Edward K. Blum · Alfred V. Aho *Editors*

Computer Science

The Hardware, Software and Heart of It

Springer

Computer Science

Edward K. Blum • Alfred V. Aho
Editors

Computer Science

The Hardware, Software and Heart of It

Springer

Editors

Edward K. Blum
Department of Mathematics
University of Southern California
Los Angeles, CA, USA
blum@usc.edu

Alfred V. Aho
Department of Computer Science
Columbia University
New York, NY, USA
aho@cs.columbia.edu

ISBN 978-1-4614-1167-3 e-ISBN 978-1-4614-1168-0
DOI 10.1007/978-1-4614-1168-0
Springer New York Dordrecht Heidelberg London

Library of Congress Control Number: 2011941144

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Contents

Part I

1 **Introduction and Prologue** 3
Edward K. Blum

2 **Computation: Brief History Prior to the 1900s** 11
Edward K. Blum

3 **The Heart of Computer Science** 17
Edward K. Blum

4 **The Software Side of Computer Science – Computer Programming** 53
Edward K. Blum and Walter Savitch

Part II

5 **The Hardware Side** 71
Edward K. Blum

6 **Operating Systems (OS)** 97
Edward K. Blum

7 **Computer Networks** 105
Fan Chung Graham and Edward K. Blum

8 **High Performance Computing and Communication (HPCC)** 139
James M. Pepin

9 **Programming for Distributed Computing: From Physical to Logical Networks** 155
Christian Scheideler and Kalman Graffi

10 Databases 169
Michael Benedikt and Pierre Senellart

11 Computer Security and Public Key Cryptography 231
Wayne Raskind and Edward K. Blum

12 Complexity Theory 241
Alfred V. Aho

13 Multivariate Complexity Theory 269
Michael R. Fellows, Serge Gaspers, and Frances Rosamond

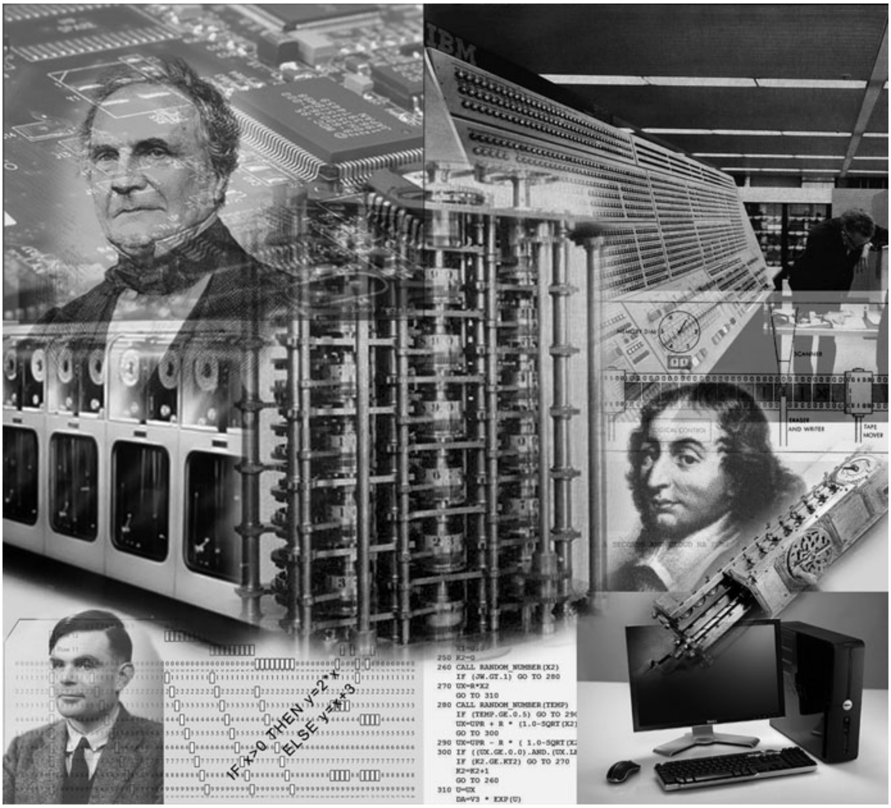
14 Quantum Computing 295
Todd A. Brun

15 Numerical Thinking in Algorithm Design and Analysis 349
Shang-Hua Teng

16 Fuzzy Logic in Computer Science 385
Radim Belohlavek, Rudolf Kruse, and Christian Moewes

17 Statistics of the Field 421
Frances Rosamond

Epilogue 467



```
    X1=0.0
250 K2=0
260 CALL RANDOM_NUMBER(X2)
    IF (JW.GT.1) GO TO 280
270 UX=R*X2
    GO TO 310
280 CALL RANDOM_NUMBER(TEMP)
    IF (TEMP.GE.0.5) GO TO 29
    UX=UPR + R * (1.0-SQRT(X2)
    GO TO 300
290 UX=UPR - R * ( 1.0-SQRT(X
300 IF ((UX.GE.0.0).AND.(UX.L
    IF (K2.GE.KT2) GO TO 270
    K2=K2+1
    GO TO 260
310 U=UX
    DA=V3 * EXP(U)
```

Contributors

Alfred V. Aho Columbia University, New York, NY, USA

Radim Belohlavek Palacky University, Olomouc, Czech Republic

Michael Benedikt University of Oxford, Oxford, UK

Pierre Senellart Télécom ParisTech, Paris, France

Edward K. Blum Department of Mathematics, University of Southern California, Los Angeles, CA, USA

Todd A. Brun University of Southern California, Los Angeles, CA, USA

Fan Chung Graham University of California at San Diego, La Jolla, CA, USA

Michael R. Fellows School of Engineering and Information Technology, Charles Darwin University, Casuarina, Australia

Serge Gaspers Institute of Information Systems, Vienna University of Technology, Vienna, Austria

Rudolf Kruse Otto-von-Guericke University, Magdeburg, Germany

James M. Pepin Clemson University, South Carolina, USA

Wayne Raskind Arizona State University, Tempe, AZ, USA

Frances Rosamond School of Engineering and Information Technology, Charles Darwin University, Casuarina, Australia

Shang-Hua Teng University of Southern California, Los Angeles, CA, USA

Christian Scheideler Department of Computer Science, University of Paderborn, Paderborn, Germany

Kalman Graffi Department of Computer Science, University of Paderborn, Paderborn, Germany

Walter Savitch University of California, San Diego, La Jolla, CA, USA

Christian Moewes Otto-von-Guericke University, Magdeburg, Germany

Part I

Chapter 1
Introduction and Prologue

Edward K. Blum

It is said that *the past is prologue*. This saying is borne out by most of the history of science, but it applies only loosely to computer science, since this "modern" science does not have that much of a past compared to the traditional sciences like physics and chemistry. In the latter, in any era the events of the past do presage the ongoing and future development of these sciences. The development of computer science proceeds at such a rapid and frenetic pace that the boundary between past events and the current state of the subject is somewhat blurred. In this book, with its advisedly provocative title referring to the *heart* of computer science, we take the position that its prologue is indeed rather brief and somewhat diffuse. To recount how the history of computer science interacts with its current state we adopt a simplifying assumption about its early history, the reasonableness of which we shall defend, to the effect that the early and defining history of computer science (the prologue) can be encapsulated in the work of one man, the British mathematician Alan Turing. Except for some very early and rather naïve contributions by the outstanding mathematicians Pascal and Leibniz and some later more substantial ones by the polymath Babbage and the logician Godel, the subject which we now call *computer science* sprang largely from the brilliant and original researches of Turing, as we shall attempt to illustrate.

Turing was undoubtedly a genius in both the mathematics of computing as embodied in his studies of what we now call *software* and in the engineering side of computing which we usually refer to as *hardware*. As this book will argue, these seemingly disparate sub-disciplines of computer science, *software* and *hardware*, are intrinsically related, although in many school curricula there is an explicit marked separation and the single subject is organized for pedagogical and research purposes into two subjects: *computer science and computer engineering*.

E.K. Blum (✉)
Department of Mathematics, University of Southern California, Los Angeles, CA, USA
e-mail: blum@usc.edu

E.K. Blum and A.V. Aho (eds.), *Computer Science: The Hardware, Software and Heart of It*, DOI 10.1007/978-1-4614-1168-0_1,

To stress this apparent dichotomy in the structure of computer science is not only unnecessary but misleading. It was not the way Turing viewed his research projects and it is counter to our view of the subject, which recognizes the intrinsic interrelationships of many aspects of software and hardware. We regard computer science as a Union (with a capital U) of its two major constit-uents, *hardware* and *software*. Admittedly, this Union is diverse and rapidly evolving and defies precise characterization. Nevertheless, in this book we endeavor to present a report on the state of this Union as of the year 2011. It is not feasible to cover all facets of this state in a relatively short book. So we shall cover a reasonably large number of its key elements which in our opinion are not only having the greatest impact on the further development of computer science itself but also a measurable impact on the surrounding world with which computer science is now so actively engaged.

We alert the reader to our style of presentation whereby we report on the state of the Union on two coordinated levels of exposition:

1. An informal intuitive level which minimizes the mathematical and technical details and aims to be readable and comprehensible, for the most part, by any intelligent layperson reader and in its entirety by a layperson reader with say a college education but one that need not include much science;
2. A concomitant technical/mathematical level that aims to provide a deeper quantitative understanding for the scientific/engineering-trained reader.

Although these two expositions could be quite disparate, in this book they are carefully coordinated so that they do not constitute two books but are united as one. We are well aware of the difficulties and hazards in such a two-level exposition; trying to produce both a "popular" book and a "technical" book between the same covers. To smooth the reader's progress we have used various literary devices, such as separating the intuitive concrete ideas from the more abstract ones by packaging the latter into appendices. The appendices are included in the main text at points close to the intuitive discussions (see the Table of Contents) and we attempt to use the same linguistic constructs for related intuitive and abstract ideas. The choice of familiar colloquial non-quantitative language, ordinary words and phrases, for unfamiliar quantitative concepts is quite common in traditional science. For example, physicists speak and think freely of electromagnetic *waves*, *interference* of such waves, inertial *forces*, *conservation* of energy, *spins* of particles, *black holes*, *time warp*, *curvature* of high-dimension *surfaces* and *spaces* and many other geometric entities. The ordinary meanings of these words and phrases carries over to the abstract concepts they denote and creates an entry into the technical contexts of these concepts. The overall exposition is constructed so that the reader can ignore or scan quickly the appendices and ultra-technical contexts on a first reading, obtaining an approximate intuitive understanding of the technical concepts and then return later to read these passages at a more leisurely pace.

The chapters are ordered so as to facilitate this two-level discourse. They start with some history and soon relate how abstract concepts (by Church and Turing) were the starting points of many practical concrete ideas. Toward the middle of the book, around Chap. 8, we begin to bring into our report of the state of the Computer

Science Union the more recent technological advances such as computer networks (the Internet and world-wide- web www), high performance computing (HPCC) by clusters to achieve super-computer power by distributed computing, large databases, secure computing (for modern banking) by means of public key cryptography, quantum-mechanics-based computing and fuzzy logic. The heart of the Union is in Chap. 3, which introduces concepts like unsolvability and Church-Turing computability and in Chap. 12 on complexity theory of solvable problems and in Chap. 13 on the recent multi-variate complexity adjunct. In prior reports of this type there were accounts of the numerical analysis progress with computation algorithms. In this book, the numerical side of computer science is represented by the single Chap. 15 on "numerical thinking". The reader is forewarned that this chapter is highly technical, treats several new topics and contains a minimum of intuitive-level discussion. It takes the reader to the outer limits of computer science and may perhaps be reserved for a second reading of the book. Similarly, at the outer boundaries, a different perspective of the logic behind computer science is presented in Chap. 4 on fuzzy logic. We end the book with some interesting real-life statistics of the field.

The state of the Computer Science Union, a dauntingly diverse and dynamic entity, cannot be assessed completely. As previous attempts to report on the Union, we can cite first the **long** (900 pages) Computer Science and Engineering Research Study (COSERS) – with the restricted theme "What can be automated" – published in 1980 by MIT Press and having engaged 80 contributors supported by the National Science Foundation and second, the **short** 200-page book "Computer Science: reflections on the field and from the field", prepared by a committee convened by the National Research Council and published in 2004.

These joint-effort reports are indeed the long and short of it in past decades. Our book is likewise the product of a joint effort and the result of collaborations by the co-authors listed prior to this Introduction in the Table of Contents. To provide compatibility and continuity across Chapters on the variety of topics a rather light editing process was administered by E.K. Blum and Al V. Aho. As a report on the state of the computer science Union, this book's objective differs from those of the COSERS and Computer Science "reflections" books. Of course, as they did, we strove herein to bring the report on the state of the Union up to date by covering significant new and recent developments. Many of these were not covered in any depth, if at all, in the two cited reports. But most importantly, we pursued the objective, dictated by our thesis that hardware and software are two sides of the same Turing-minted computer coin, of presenting Computer Science as a rational Union of its many diverse elements. We tried to avoid a heterogeneous disconnected collection of essays. We used Turing as a unifying figure.

Turing studied and worked primarily at Cambridge University in the 1930s and 1940s. His biography, called "Alan Turing the Enigma", (Simon and Schuster Inc, 1983) is a prodigious work by Andrew Hodges. It gives us an intimate view of Turing as a human being and as a scientist. As one works through multiple re-readings of "Turing the Enigma" and Turing's research papers (see Chap. 3) and many other research papers, the following two *synopses* of items take shape and they will be mentioned and discussed in various chapters of this book. After much consideration of these items, we came to think of computer science like Turing did, as progressing along two parallel concurrent paths: Hardware (i.e. machines) and Software (i.e. programming). This suggested a reasonable way to organize this book, that is, as already remarked, the book traces the interlocked development of both Hardware and Software as two faces of the same computer science "coin". The reader will find that the chapters touch on the following Hardware topics among others.

Hardware Synopsis

The book mentions, but skips details of the Babbage "engine" and quickly jumps to the 1940–1950s. That is when our modern Computer Science story begins. Among the hardware developments the book selectively mentions and discusses (sometimes very briefly) the following bits of history in the period 1940–2010:

1. Small relay-based "office" or "business" machines produced as standard products by IBM for business data processing were adapted to numerical scientific computation by being programmed by inserting math-level instructions on stacks of punched cards. These were the CPC machines or Card Programmed Calculators. They were awkward to use, slow and small-scale but they worked better than the ubiquitous electromechanical desk calculators provided by Marchant (and others).
2. Professor Aiken at Harvard built a large-scale relay machine called the Mark 1 which ran under control of programs on punched paper tape. Like all relay-based machines, it was slow and not very successful (as was one attempted by Turing who actually fabricated his own relays!).
3. To the rescue came the electrical engineers Eckert and Mauchly (U. Penn) who with support from Sperry-Rand built the Univac out of hundreds of vacuum tubes. It took up an entire room but was the first commercially successful "electronic" computer. Sperry-Rand was an IBM competitor for business applications.
4. Not to be outdone, IBM produced a competitor electronic machine called the 701. Input/output was on large magnetic tape drives and still some punched cards since IBM "owned" the Hollerith cardpunch machines, card readers and sorters and other "business" machines and stacks of 80-column cards could be easily stuffed into card-readers attached to the ALU (arithmetic-logical unit) of a 701.The big-company battle was joined. The 701 was quickly followed by the IBM704 , which competed with the impressive new Sperry 1101. These were transistorized and fast machines, say running at kiloflop/s speeds (flop = floating point arithmetic operation) and required large air-conditioned installation spaces for the racks of electronic circuits and tape drives.
5. In England, Manchester University engineer Williams built an electronic machine (The engineers had taken the lead here.). Other European countries

like France and Germany were effectively out of the running or else "occupied" by I (for international) BM.

The preceding brief item 5 is somewhat inaccurate. Sir Maurice Wilkes in an interview excerpted in CACM 09/2009 vol 52 No. 9 recalls that he was in charge of developing the EDSAC electronic computer at Cambridge University during 1945–1949. At about the same time, the group at Manchester University led by Freddie Williams built a competitor computer. It seems that the main focus of these two English groups was on the computer memory device. Wilkes favored mercury delay lines with data stored as traveling acoustic pulses (non-electronic!) and Williams promoted cathode ray tubes (CRT) with memory consisting of continually refreshed electrostatic storage of charge patterns on the CRTs. The so-called "Williams tubes" won out in follow-on machines built by IBM and RCA until replaced by magnetic core memories invented at MIT. The main point to be highlighted in the English Hardware development exercises is that both machines used the von Neumann (v.N) architecture as their basic overall system design. In the Wilkes interview, he states very clearly that he had access to and had read the famous report by John v.Neumann (mathematician), H.H. Goldstine (mathematician) and Arthur Burks (logician) that laid out in great detail the "v.N architecture" for a new machine, the ENIAC, to be built in the U.S. for the military. It is somewhat amusing to read Wilkes disparage v. Neumann by claiming that "of course, he rather despised engineers … although he got on with them." This is part of what Wilkes apparently views as a fundamental disagreement between mathematicians with their absorption in Software and engineers with their absorption in Hardware. Actually, v. Neumann, though a mathematician, was a multi-faceted genius who knew the relevant Hardware physics, electronics and engineering, including the important Boolean algebra application to electronic circuit design (initially considered a "daft idea" by engineers according to Wilkes) as a tool for designing switching circuits. Here we see the early interplay of Hardware and Software in many of the ideas on the Hardware side of computer science (e.g. switching circuits) that sprang from the Software side; e.g. from the mathematics of Boolean algebra as presented in Chap. 3 Appendix G. This is a fascinating theme in the state of the Union which our book explores in depth.

6. The "big-machine" era climaxed in the 1960s with victory for IBM and its 709, then 7090, then 360 fast reliable machines. Sperry gradually bowed out. Burroughs Inc. competed briefly with its B5000. DEC (Digital Equipment Corp) also competed with its medium-sized *minicomputer* PDP-8.
7. Honeywell-CDC made some important inroads with its Control Data machines (CDC 6600). The other big company competition was the innovative Cray company whose design genius Seymour Cray (having left CDC) introduced radical new computer architecture designs with multi interacting ALU units. The Cray's were the first "super-computers". They were capable of doing multi megaflops/s

8. The scene changes in the 1960s and thereafter. At the opposite end of the size spectrum, enter the personal computer (PC) introduced by "big blue" IBM. The PC's novel architecture involved a *mother* board and other printed circuit boards as components that could be easily assembled as independent "chips". (See Chap. 5 Appendix.) The PC became a practical and widely adopted computer when Bill Gates founded Microsoft, which provided the practical DOS operating system for PC's. This key Software concomitant was not attended to by IBM and Bill Gates (a Harvard dropout) became rich. Very soon IBM also lost its Hardware lead to Intel which provided very fast printed circuit boards as off-the-shelf chips to many companies that could produce PC's using DOS and undersell IBM. Intel remains a leading computer company today.
9. The Apple Computer, allegedly first built in a garage by two Stanford students (Steve Jobs and Wozniak), added to the Silicon Valley Hardware explosion. DEC and Apple developed a clever "windows" display system employing a "mouse" as part of the user interface of PC's (Chap. 5 Appendix 2) and this was "taken over" by Microsoft for its new Windows operating systems using the mouse and monitor displays. It has become the user physical interface of choice.
10. Clusters of PC's operating in parallel in connected local networks of computers became the new super-computers running at gigaflop and some at teraflop (10^9) speeds. (See Chap. 8).
11. Proponents of new systems, called "cloud" computers (Chap. 8), envision connecting large populations of PC's in networks that can utilize thousands of PC's cooperating on large-scale computations. Another radically new idea now being tested is the quantum computer based on quantum mechanics at the atomic level (Chap. 14). This idea was one of many imaginative proposals by the Nobel-prize-winning physicist Richard Feynman, who was an early pioneer in computing at the Los Alamos atomic bomb project (along with von Neumann).

Among the Software topics covered in the book are some which we now list in another synopsis. Some readers will observe and hopefully excuse our failure to cover many other Software history items worthy of mention.

Software Synopsis

The word *software* came into existence around 1950 as a counterpoint to the engineering usage of the standard word "hardware". While it usually refers to programming languages and concepts, in this book we use "software" in a broader sense to also include the mathematical theory underpinnings of computer science. Restricted to a purely programming context, here is a list of software achievements taken from the Patterson-Hennessy book (PH) listed below (and reflecting their opinions).

1. 1954 **Fortran**, John Backus, Fortran I, II, IV, 77, 90 for the IBM 704 computer
2. 1958 **Lisp**, John McCarthy
3. 1960 Algol-60
4. 1960 Cobol
5. 1968 Pascal, Nicholas Wirth
6. 1968 C language, Dennis Ritchie
7. 1967 Simula-67, O-J Dahl and K. Nygaard
8. 1970s, Smalltalk, Xerox PARC
9. 1970s, CLU
10. 1980s, C++
11. 1990s, Java

We shall limit our coverage of programming languages in Chap. 4 to software items 1, 6, 10 and 11 in this list, since this will suffice to illustrate the major ideas in computer programming. Some further discussion of item 7 will be found in Chap. 9.

Some General Remarks on the Nature of Computation

Theoretical computer science is a broad research area which we cover as part of our software exposition. Our Chap. 3 recounts Turing's research on computability by Turing machines and includes an appendix on Church's lambda calculus which has had some impact on programming languages.

Chapter 3 introduces the *Church-Turing thesis* which postulates that Turing machines and the equivalent Church lambda calculi define the essence and totality of *computation*. This thesis has the somewhat disturbing consequence that certain innocent-sounding problems in computation are *unsolvable*. The famous Godel *undecidability* results which show that the standard applied predicate logics are *incomplete* and therefore that the whole formalist program to mechanize mathematics is unachievable is presented in Appendix G of Chap. 3. Speculations are presented that other approaches to *computation* such as quantum computers (Chap. 14) may circumvent some unsolvability issues. However, these are just speculations which have yet to bear fruit. The inherent *complexity* of problems in the Church-Turing universe is expounded in Chap. 12. Chapters 3, 14 and 15 provide a framework for predicting the progress and limitations of computer science as seen from its current state.

The book does not cover an area that was of interest to Turing and is pursued by current researchers: *artificial intelligence (AI)*, although the fuzzy logic chapter has a few comments on AI. The book ends on a rather sober note with Chap. 17 on statistics of the field. These could be the basis for some predictions but we resist the temptation. In particular, except for some tantalizing remarks in Appendix G (Chap. 3), the book takes no position on the future shape of the computer science Union. In contrast, a recent article in Time magazine (Feb. 21, 2011) reports on forecasts of amazing progress to be expected in AI as computers increase in speed. The article reports on conjectures that computers will be able to outthink humans in about 2045.

These conjectures fail to realize that the human brain is not a fast device. Neurons fire voltage spikes at quite slow rates. So speed is likely not the issue in human intelligence. Rather it is the size and complex structure of the brain, produced by thousands of years of evolution resulting in billions of interconnected neurons that may underly human consciousness and thinking. Furthermore, as the chapter on quantum computers suggests, there may be physical limits to electromagnetic circuit speeds, such as the limiting speed of light for signal transmission and quantum mechanical effects for microscopic circuit elements. Whether quantum mechanics provides a possible alternative to electromagnetic circuits remains an open question. These are fascinating issues in the state of the Computer Science Union which our readers will encounter and we hope will enjoy contemplating.

Reference

Computer Organization and Design, The Hardware Software Interface by David Paterson & John Hennessy, Morgan Kaufman 2005 Page 2.19-5ff A Brief History of Programming Languages

Chapter 2
Computation: Brief History Prior to the 1900s

Edward K. Blum

Computation, as a human activity, has a long history extending back to such ancient civilizations as Egypt, Sumeria (Babylon), Assyria, Greece, and Rome and the later civilizations of medieval Europe. It was used in commerce, agriculture and astronomy. However, it was not an activity of the common man in the street.

Egyptian mathematics is known partly from studies of the large Rhind papyrus, which is possessed by the British museum (and a small piece of which is in the Brooklyn museum according to Carl Boyer's "A History of Mathematics", John Wiley 1968, which we refer to as a source for some of the history which we summarize below). Despite their conjectured computational prowess in building the quite perfectly shaped pyramids, Egyptian mathematicians subsequently dropped behind their Sumerian colleagues in capabilities. This may have been due to the awkward and inelegant hieroglyphic notation for Egyptian numerals.

Sumerians lived in the region known as Mesopotamia, the fertile valley between the Tigris and Euphrates rivers which is now Iraq. Their principal city Babylon, of biblical renown, gave its name, Babylonia, to their culture and civilization which flourished from about 2000BCE to 600BCE. The Sumerians developed the earliest form of written language. It comes down to our attention in the guise of thousands of preserved clay tablets on which are carved symbols in cuneiform (wedge-shaped) patterns. Some of these symbols are numerals denoting natural numbers (positive integers) and were involved in practical computations for agriculture and business. Many tablets studied by archeologists in records of the Hammurabi dynasties, 1800–1600BCE, exhibit number systems such as the common base 10 numerals and an unusual one utilizing base 60 numerals in their astronomy. Base 10 numerals were used in daily transactions. In fact, the present-day positional notation for decimal numerals, where position of a digit is determined by a power of 10, was in use by the Babylonians. Its facilitation of numerical computations like addition

E.K. Blum (✉)
Department of Mathematics, University of Southern California, Los Angeles, CA, USA
e-mail: blum@usc.edu

E.K. Blum and A.V. Aho (eds.), *Computer Science: The Hardware, Software and Heart of It*, DOI 10.1007/978-1-4614-1168-0_2,

is familiar to us. An important symbol in positional notation, the zero, was not initially available but was eventually invented by the Babylonians. Their more complex computations beyond addition included special tables for multiplication. Given this fairly sophisticated state of numerical computation in ancient Babylonia it is surprising that later civilizations did not develop more complex concepts for computations.

As the center of ancient civilization slowly moved toward the Grecian cities and lands on the Mediterranean sea, Babylonian mathematics moved with it. It was taken up by two Greek groups, one led by Thales and the other by Pythagoras. The Pythagoreans practiced various nonconformist cult-like philosophies. Although they developed new mathematics for geometry, they did not do much to advance computation methods. They coined the words "philosophy" to describe "love of wisdom" and "mathematics" to describe "that which is learned". As this indicates, they regarded mathematics as a much broader intellectual activity which emphasizes love of wisdom rather than practical computational goals. Nevertheless, the Pythagorean motto is said to have been "All is number.", which may reflect the influence of the Babylonians who attached numerical measures to compute almost everything, from the motions of heavenly bodies to the values of their slaves.

As opposed to the Pythagoreans, the growing society of ordinary Greek citizens was a society of shrewd traders and business men and their needs were satisfied by a fairly low level of computation. They used two numeral notations for the integers, the more primitive one resembling the later Roman system with a special symbol for the number 5. We know the Roman numeral notation is less suitable for computation than the positional decimal notation. Both Greek numeral systems were weak in the way they represented fractions. Decimal positional notation for fractions was rarely used by the Greeks or other societies until the Renaissance.

Around 600–400BCE deductive methods were introduced into mathematics and adopted later by Euclid in his *Elements* books on geometry. This was the age of Plato and Aristotle. Deductive computations were highly prized by the Platonic school. In his book *The Republic*, Plato states that *arithmetic* theory, by which he meant deductive proofs, is superior to computation (called *logistic*) as an intellectual pursuit. The Platonic school grappled with numbers like $\surd 2$ which they proved to be irrational (not a ratio of 2 integers m/n.) (Remember how? Hint: use contradiction after supposing $\surd 2 = \mathrm{m/n}$. Then $\mathrm{mm} = 2\ \mathrm{nn}$. Then mm has an even number of two factors, whereas 2 nn has an odd number, which is a contradiction).

The Greek empire was split into several pieces when Alexander the Great died. In about 300BCE, the Egyptian part was under control of Ptolemy I. He established an outstanding school in Alexandria with a great library, world-class scholars and teachers. Among the latter was the mathematician Euclid who is the author of the famous textbook the *Elements*. The first 13 books, or chapters, of the Elements are devoted to plane geometry and the next three to number theory. As taught in secondary schools today the true statements in the geometry books of the Elements, called *theorems*, are proved by deduction from a few postulates, or *axioms*. We shall see that the deductive steps make these proofs a kind of non-numerical computation and further, this served as an example of the logical proofs advocated in the 1900s as a means to derive all of mathematics (see Chap. 3 Appendix G). The Elements were translated into Arabic, then into Latin in the twelfth century and finally in the sixteenth century into various European languages. It has appeared in a large number of editions, a number perhaps only exceeded by the Bible.

Ancient India and China both had number systems for computation. Chinese numerals were mainly decimal, the individual digits d from 1 to 9 being denoted by a multiplicity of d strokes, or "rods". A positional notation for the rods allowed the invention of counting boards as primitive computers. The word *abacus* for these devices may originate from the Semitic word *abq*, referring to the sand tray used as a counting board for the rods in other lands as well as China. In Arabia, the abacus had ten balls per position wire. The Chinese had five balls on upper and lower wires separated by a bar. The Chinese were also familiar with computations on fractions and negative numbers. By about 300BCE, the Indian notation of individual ciphers for the digits 1–9 had evolved to the Hindu numerals which we use today and it was recognized that they can be used in all decimal positions.

In about 800AD, a 100 years after the founding of the Muslim empire by Mohammed, there was an awakening in Arab countries to science and mathematics. A university comparable to the one in Alexandria was established in Baghdad. Around 850AD, a mathematician on its faculty named Mohammed ibn-Musa al-Khowarizmi became so well- known for his published works employing Hindu numerals that we now refer to them as being Hindu-Arabic in origin. As a corruption of his name, his rules for operating on these numerals became known as *algorithms*, a word now applied to any computation method specified by a systematic sequence of well-defined rules. Also from his work called Al-jabra wa'l muqabalah came the modern word *algebra* and knowledge of that subject as of that era was made available in Europe. In Persia, around 1050–1123AD a book on algebra was published by the mathematician Omar Khayyam, better known in the west as a poet. This book treated computation of solutions of quadratic equations and gave geometric solutions of cubic equations, which we now know can be generally solved by purely algebraic formulas.

Europe in the middle ages did not experience great progress in mathematics or computation but relied on classical ancient Greek knowledge. In the Renaissance period, 1400–1600, the main mathematical trend was in algebra. The Italian mathematician Geronimo Cardano, known to us as Cardan, published works on the solution of the cubic and quartic equations actually discovered by others (Tartaglia and Ferrari). In the modern period which followed, Galileo Galilei (1564–1621) and B. Cavalieri (1598–1647) and Johann Kepler (1571–1630) developed mathematics and computation applied to the physical world. Francois Vieta (1540–1603) worked in algebra. John Napier (1550–1617) of Scotland and Henry Briggs (1561–1631) of England created logarithms as a means of computation of multiplication more easily.

The center of new mathematical discovery moved from Italy to France, where it was dominated by Rene Descartes (1596–1650), Pierre Fermat (1601–1665) and Blaise Pascal (1623–1662). Descartes's algebraic notation for his published mathematics created a modern expository style wherein letters at the beginning of the alphabet denote parameters, letters near the end denote unknown quantities, superscript exponents denote powers and + and − the usual positive and negative quantities.

At this juncture in our brief tour of the history of computation, when we consider Pascal, we come upon an actual computer device (We ignore the abacus as too primitive). Pascal had wide-ranging interests in mathematics and at the age of 18 he planned a computer device which he actually built and of which he sold about 50. It was mechanical in its usage of gears. Again, when we consider the work of the mathematician Leibniz, who is credited along with Newton with the creation of the Calculus, we learn that Leibniz invented a mechanical computer device, called the *stepped reckoner* which was based on a stepped drum mechanism and an intricate gear-work mechanism. Leibniz built a wooden model which he brought to London in 1676. In principle it could perform all four arithmetic operations on integers, whereas Pascal's machine could only add and subtract. According to a Wikipedia encyclopedia article, its design was beyond the mechanical fabrication technology of that time and there were design flaws in the positional carry mechanism (always a challenge in computing machines). These factors made its operation unreliable. However, the stepped wheel device, called a *Leibniz wheel*, was employed in many computer devices for 200 years, even in the 1970s in the Curta hand-held calculator. Mechanics was the chief mode of fabrication of computing machines in the ages before electronic devices. Leibniz, a polymath, lived before the advent of electric motors and therefore his was a brave hand-operated attempt to mechanize computation. According to Wikipedia, there is a 16 digit prototype which survives in Hannover. It is about 67 cm (26 in.) long and consists of two parallel parts, an accumulator which can hold 16 digit results and an eight digit input section having eight dials and a hand crank which is turned to cause operations to be performed.

Leibniz, like mathematicians of his era, was also a physicist. As such he developed a theory of kinetic energy (mass times velocity squared) discovered by his mentor Huygens. Leibniz believed kinetic energy was a more fundamental physical quantity than momentum (mass times velocity), which Descartes and English scientists regarded as the fundamental quantity. Leibniz proposed a conservation of (total) energy law but it was based on metaphysical grounds rather than engineering facts. Among Leibniz's other discoveries was the principle of separation of variables for solving certain partial differential equations. Further, he used determinants long before Cramer did. Despite his accomplishments in theory, he was also an advocate of applied science and invented many devices such as wind-driven propellers and water pumps, mining machines, hydraulic presses, lamps, and clocks. His stepped wheel computer was only one of his mechanical inventions.

These historical examples of computing machines were the only notable ones until the year 1822 when the Cambridge mathematics professor Charles Babbage, a polymath like Leibniz, invented a mechanical computer, called the Difference Engine. This was a special-purpose computer for computing polynomial values by finite differences, polynomials being good approximations to the functions needed to calculate astronomical tables, the main objective. Babbage had difficulty obtaining funding to build a Difference Engine. A few difference engines were built by one Per Scheutz in about 1855. The second Difference Engine built had 8,000 parts, was 11 ft long and weighed 5 t.

Understandably, a general-purpose computer called the *analytical engine* was later designed by Babbage in 1837 and he worked on its development until his death in 1871. It was never built, for political and funding reasons. But many of its design features were implemented in modern computers. Its data and programs were input by punched cards in the manner already employed in controlling Jacquard looms. Output of intermediate results was also on punched cards and final output was to be by printer. Ordinary base-10 arithmetic was used internally. There was a memory of capacity of 1,000 50-digit numbers. Analogous to the *central processing unit* (*CPU*) in a modern computer, the mill (*arithmetic unit*) had its own internal built-in operations, *microcoded* by pegs inserted in a drum. The programming language was similar to a modern *assembly language*. It provided for *loops* and *conditiona*l *branching* (see Chap. 4 on Software). An Italian mathematician whom Babbage had met while traveling in Italy wrote a description of the programming language in 1842 and it was translated into an English version in 1843.This was read by Ada King, Countess of Lovelace, Byron's daughter, who was herself a gifted mathematician. She added to the English version and actually wrote specific programs. For this reason she has been called the first programmer and a recent language was named ADA in her honor. Babbage died in 1871, unable to get funding to build his analytical engine. In 1910, his son reported that a part of the engine (the mill and printer) had been built and used successfully and he proposed to build a demonstration version with a small memory having 20 columns with 15 wheels in each. Closely related electromechanical (relay) machines were later worked on by George Stibitz at Bell Laboratories and Howard Aiken (the MARK I) at Harvard. Aiken attributed much of the MARK I design to Babbage's Analytical Engine design (see Chap. 5 on the Hardware side of Computer Science for the modern history of computing machines).

It is of some interest to remark that many of the key players in the history of Computer Hardware, Pascal, Leibniz, Babbage, Turing and von Neumann were equally adept at theory and were in fact polymaths of genius stature. They contributed to many other disciplines.

Babbage was the Lucasian Professor of Mathematics at Cambridge (the Chair once held by Newton), was a founder of the Royal Astronomy Society, worked in cryptography (like Turing) where he broke what is known as Vignere's autokey cipher to aid the British military, invented the "cowcatcher" device to clear the track in front of railway locomotives, invented a medical ophthalmoscope which was later re-discovered by Helmholtz, and, again like Turing, exhibited several eccentricities. For example, having read the poet Tennyson's lines "Every moment dies a man, Every moment one is born", Babbage contacted the poet to point out "if this were true the population of the world would be at a standstill… in truth the rate of birth is slightly greater than death… so your lines should read "Every moment dies a man, Every moment 1 1/16 is born." which is sufficiently accurate for poetry." Babbage is commemorated in several ways: the crater on the moon called the Babbage Crater, the Charles Babbage Institute at the University of

Minnesota, the Babbage lecture hall at Cambridge, and a railway locomotive named after him by British Rail. His programming colleague was Ada Augusta Byron, the only legitimate child of the great poet George Gordon, Lord Byron. Her mother took her away as a child from Byron and raised her to study math and science. She married William King who became Earl of Lovelace and she became Countess Lovelace. They had three children. As noted above, she translated into English the French version of the article on the Analytical Engine written by the Italian engineer Manabrea (who later became a prime minister of Italy). She added many of her own notes and wrote a program for the engine to compute Bernoulli numbers. She died in 1852 and is buried next to Lord Byron.

From the preceding brief account of the early history of computation, we learn that, except for a few explicit examples such as the invention of logarithms by Napier and Briggs and the rather primitive computers invented by Babbage, Leibniz and Pascal, in modern civilizations the ideas involved in computation were not developed much beyond their ancient forms. By contrast with its rather sparse history prior to 1936, the almost all-pervasive presence of computation in today's world is a consequence of the recent and rapid growth in only the last 75 years of the *hardware* and *software* sides of the modern subject known as *Computer Science*. If modern man wishes to understand and cope with the world he lives in, he should have some working knowledge of Computer Science. This book is intended to impart such knowledge. To keep this knowledge accessible and within reasonable and readable bounds, the book does not attempt to give a complete account of the development of Computer Science nor an encyclopedic coverage of it. Rather it presents the main ideas of Computer Science in a hopefully comprehensive and coherent fashion. It does this at several levels, one given in expository sections at a rather intuitive and easily understood level not requiring prior advanced education and other levels in sections, usually Appendices, requiring some advanced prior education, say at a college level. The reader is advised to journey through the various chapters at a leisurely pace and choose to skip over the advanced sections and Appendices at first and perhaps return to them in a second more strenuous reading. Our authors wish you Bon Reading Voyage.

Chapter 3
The Heart of Computer Science

Edward K. Blum

The active beginnings of modern Computer Science are somewhat diffuse and even controversial but, as we shall show, it is reasonable to consider the major active beginnings of Computer Science as being rooted in the works of one man, Alan M. Turing, starting with his epoch-making 1936 paper, On Computable Numbers with an Application to the Entscheidungsproblem. Proceeedings of London Mathematical Society, ser.2, vol. 42. 1936, corrections ibid. vol.43, 1937. This paper has been re-published by Dover Pub. Co. in the 2004 anthology *The Undecidable*, edited by Martin Davis. Some may not completely accept our opinion that Turing was the single fountainhead of the main ideas of computer science. However, one cannot fail to be impressed by the prescient quality of his wide-ranging research as displayed in his published papers and reports. At the very least, he was a leading thinker of the new discipline of computer science and contributed to both its hardware and software content.

Turing was an English mathematician, and still a student at Cambridge University when he wrote the above paper. Unlike the early historic interest in computation (see Chap. 2) on the part of many scientists focusing on the practical aspects of ordinary numerical computation, Turing's main interest in computation arose quite differently from a theoretical aspect embedded in an abstract profound mathematical problem called the Decision Problem. This problem was designated by its German mathematician originator, David Hilbert, as the Entscheidungsproblem (Decision Problem), as in the title of Turing's paper. Although this may be premature for some readers, we shall now give a background discussion of the Decision Problem, since this will set the stage for Turing's role as the principal founder of Computer Science and prepare us for some of the controversies that have plagued the subject. To be comprehensible at an intuitive level, the explanation of the Decision Problem in this early chapter will necessarily be an oversimplification

E.K. Blum (✉)
Department of Mathematics, University of Southern California, Los Angeles, CA, USA
e-mail: blum@usc.edu

E.K. Blum and A.V. Aho (eds.), *Computer Science: The Hardware, Software and Heart of It*, DOI 10.1007/978-1-4614-1168-0_3,

of what is an abstruse profound problem. Its complete character will be amplified in Appendix 1 below. In section "The Decision Problem of Formalist Mathematics" of this chapter which follows we give a summary description of it with omission of certain distracting technical details.

The Decision Problem of Formalist Mathematics

In the early twentieth century, there was an ambitious research effort by mathematical logicians, like Russell and Whitehead, to develop all of mathematics as a *formal system* based on mathematical logic. This was also a dream of Leibniz which he could only speculate on, since in his era the appropriate formal system did not yet exist. According to this *logicistic* view and the mathematician Hilbert's *formalist* view of mathematics, all the true statements in mathematics, called *theorems*, were to be shown to be *derivable* or *provable* by purely *formal proofs*, a *formal proof* being a sequence of well-established deductive steps starting from *axioms* in a formal system of logic. For example, the reader may recall that this kind of formal proof procedure is applied in the Elements book of Euclid to elementary Euclidean geometry to derive theorems about congruent triangles. The deductive steps in a formal proof are little more than applications of logical symbol-manipulating rules. There can be no appeal to the meanings or *interpretations* of the formulas to be manipulated. For example, in a formal proof of a theorem about congruent triangles in Euclidean geometry there are no statements about the meanings of symbols and formulas for angles and sides of triangles as geometric objects. Of course, an understanding of such meanings obtained through i*nterpretations* of their symbols as denoting intuitive real geometric objects is possible and, in fact, can be very helpful in devising a formal proof.

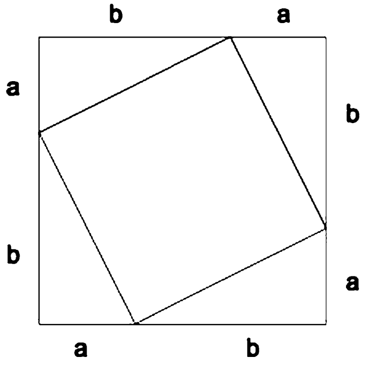
So, *formal pro*ofs are essentially *computations* which manipulate symbols and formulas according to simple and well-defined rules. This is the kind of computation that interested Turing. In the formalist view, it is reasonable to assume that a formula expressing a mathematical theorem can be proven by computations performed on symbols in a mechanized manner that makes no reference to the meanings (interpretations) of the symbols. In this view, mathematical theorem proving, a major mathematical activity, becomes a game in which symbols are manipulated in a purely mechanical manner. Many mathematicians rejected this formalist view and insisted that human intuition and insight is important and often necessary in proofs of theorems. Carried through to its ultimate conclusion (now seen as naive), in his formalist view of mathematics Hilbert preached the dogma that for any theorem there is a formal proof. In 1931, the young mathematician Godel demolished this dogma, that is, he showed it to be false (See Appendix 1 on Godel's Incompletenes Theorem below). He did so by adjoining axioms for the arithmetic of natural numbers (i.e. the non-negative integers) to the usual logical axioms (see Appendix 1) to obtain a formal system, F(N), in which the formulas can be interpreted as statements in ordinary intuitive number theory, N. Such a f*ormal number theory* F(N) is an important initial formal construct in the logician's attempt to formalize all of mathematics. Godel cleverly exhibited a true statement, U say, about natural numbers such that the formalization F(U) of U as a formula in F(N) is not formally provable by the deductive rules of F(N), nor is its formalized negation F(Not-U) (In Appendix 1, we shall give the formulation of F (U)). This showed, counter to Hilbert's dogma, that the formal system F(N) of number theory must be considered to be logically *incomplete,* since if the logical system F(N) were strong enough, as Hilbert believed it was, it is expected by the usual rules of logic that one of the formalizations, either F(U) of statement U or F (Not-U), should be provable, that is, derivable from the axioms of F(N). One of these formalized statements must be interpreted as a *true* statement under the natural interpretation in N of F(N). provided that the system F(N) is *consistent*, that is, does not contain contradictory formulas. It is generally believed that N is consistent since no number-theoretic contradictions have ever been discovered, so that F(N), which formalizes N, should also be consistent. An inconsistency in F(N) would be interpretable as an inconsistency in N. In fact, as the Godel Theorem appendix shows, U is easily seen to be true by intuitive reasoning using the obvious interpretation in N of the formalization F(U) of Godel's statement U. So, according to Hilbert's dogma, F(U) should be formally provable. Godel showed that F(U) is not formally provable, establishing the falsity of Hilbert's dogma. In fact, since F (Not-U) was also shown to be not formally provable, F(U) is called an *undecidable* formula. So the formal system F(N) is *incomplete* in the sense that it does not have sufficient proving power. The deficiency cannot be remedied by adjoining more axioms to F(N). Doing so only generates other undecidable formulas. Therefore, Hilbert's formalistic main goal of obtaining formal proofs of all of mathematics is not achievable. Godel's result destroyed much of the formalist motivation for mechanizing mathematics. It also changed the centrality of the role of formal computation in proving theorems. In hindsight, we might say that the Godel formula F(U) should perhaps not have come as a surprise, since logicians had previously encountered paradoxical statements which could neither be proved nor disproved; i.e. were undecidable. A famous example is the liar paradox contained in the statement, "I am now lying." Of course, Godel's formula U, although undecidable, is nevertheless true (See Appendix 1 which follows).

However, despite the incompleteness of F(N), Hilbert still hoped that there was a Decision (Entscheidungs) Procedure which can decide if any correctly-formed formula, F(S) say, which is interpretable in N as an intuitive mathematical statement S, is formally provable and if not whether its negation NotF(S) is provable. Such a Decision Procedure would be useful in the formalist approach to mathematics by telling a mathematician whether F(S) is formally provable or not and therefore whether one should attempt to find a formal proof or a disproof (i.e. a proof of NotF(S)). Of course, in consequence of Godel's result, if F(S) is undecidable, then the Decision Procedure would have to indicate that neither F(S) nor NotF(S) is provable. But the established incompleteness of F(N) did not preclude the existence of a Decision Procedure, which would still lend significance to the formalist view. At this juncture, enter A.M. Turing and A. Church both of whom proved that there is no such general Decision Procedure, another weakening of the formalist doctrine. We now say that the Decision Problem is *unsolvable*, another devastating blow to Hilbert's attempted formalization of mathematics. As a side remark, we might say that these proofs of Turing and Church were incidentally victories for those mathematicians who all along had questioned Hilbert's dogma and insisted on the primary importance of intuitive mathematics.

To rigorously prove the unsolvability of the Decision Problem, Turing and Church had to propose precisely what decision procedures would be encompassed (allowed) in their proofs of unsolvability. Turing's proposal for a procedure was the *Turing machine*, which was readily accepted by the formalists and then turned out to have much wider implications for computation in general. Church's proposal was the *lambda calculus,* a system for defining *effective procedures*, which are equivalent to Turing machines, as Turing showed in his 1936 paper.

For the reader's amusement and to provide a concrete example of a formal proof we consider the famous Pythagorean Theorem that states for a right triangle with sides a and b and hypotenuse c the mathematical formula $c^2 = a^2 + b^2$. We give a geometrically oriented (but still formal) quick proof here. This well-known proof consists of a diagram in which there is a square S_1 of side a + b (The existence of S_1 can be proved from Euclid's axioms). On each side of the square, mark the distance a by a point P_a. Join the four points P_a by four lines $L_1, \ldots L_4$. These lines form a second square S_2 of side c inside the first square together with four right triangles. They form the hypotenuses of length c of four right triangles with sides a and b. Prove all this formally by proving that the lines L_i meet at right angles, which follows from the geometry of the figure, since the acute angles of each right triangle add up to 90°. Then compute the area of S_1 as $(a + b)^2 = a^2 + 2ab + b^2$. The area of each right triangle is ab/2, their area sum being 2ab. The area of square S_2 must then be $a^2 + b^2$. But its area is also c^2 (QED as Euclid would say,).

We include a sketch of the diagram of the two squares below.



The Turing Machine

In 1936, there was no precise definition of an executable procedure that should be allowed in formal proofs to carry out the rules of inference of a formal system. So the concept of a Decision Procedure in Hilbert's Decision Problem was not a precisely defined concept. Yet, most mathematicians and logicians regarded Hilbert's Decision Problem as well-defined, perhaps influenced by Hilbert's preeminence as a world-renowned mathematician.

Still, for a researcher to attack the Decision Problem rigorously it was necessary that he/she have a precise definition of a procedure that would be acceptable to the formalist school. First, it should be evident that the individual basic steps of the proposed procedure can actually be carried out by a human computer or perhaps a simple machine. Furthermore, the class of procedures encompassed by the definition should include any that would likely be used to reach decisions. This latter criterion would need to be certified by the formalists, a somewhat risky requirement.

The situation remained problematic until 1936 when there came a breakthrough by the American logician Alonzo Church (Princeton University).

Church formulated a definition of what he called an *effectively calculable function* or an *effective procedure* utilizing three simple rules in a system called the *lambda calculus* (See Appendix 2 below). He argued persuasively that the lambda calculus procedures encompassed all procedures that would be admissible for the Decision Problem. He then proved that the Decision Problem is unsolvable, that is, there is no effective procedure which can decide if an arbitrary mathematical formula is formally provable.

Somewhat later in 1936, working independently and without knowledge of Church's result, A.M. Turing at Cambridge University defined a more machine-like concept of a procedure, now called a Turing Machine, that would be general enough to be applied to the Decision Problem. He confirmed Church's result that The Decision Problem is unsolvable, this time by Turing Machines.

Turing also established that Turing machines and lambda calculus procedures are equivalent, that is, the existence of one for any purpose implies the existence of the other. This led to the ***Church-Turing thesis***: all computations can be defined by either of these equivalent types of procedures.

The subsequent widespread adoption of this thesis in effect specified the subject matter of computer science: the universe of Turing machine computations. We prefer the Turing machine approach as being more intuitive than that of the lambda calculus since it combines hardware (machines) and software (machine programs) ideas. It supports our earlier contention that Hardware and Software are two connected sides of the computer science "coin".

We accept the Church-Turing thesis that all computations can be performed by Turing machines, or by lambda calculus effective procedures. The lambda calculus definition emphasizes the construction of effectively calculable functions, paying careful attention to variables as arguments of the functions. It has influenced

Software development of some programming languages such as LISP and the usage of subroutines. However, the Turing machine viewpoint is more intuitive and its consequences are more prevalent. We shall adopt the Turing viewpoint. Therefore, although the details are somewhat tedious, we need to give a description of a Turing machine to further explain its influence on Computer Science. We shall intentionally skim over many of the details found in Turing's 1936 first paper so as to avoid, or at least reduce, the monotony of such a description. We recognize that in the era of the creation and promulgation of the Turing machine concept by Turing and others, 1936–1950 say, there were very few, if any, real Hardware computers available to computer scientists, engineers and mathematicians. Computers and similar technological instruments were unfamiliar in that early computing milieu and concepts which are today quite familiar were then new and therefore required detailed explanations.

The Turing Machine

Turing's 1936 paper on Computable Numbers and his subsequent reports on real computer design, such as the ACE computer project, are heavy with what we now regard as obvious concept details. In the following exposition of Turing machines, we shall omit these details on the assumption that they are indeed familiar to modern readers who use laptops, cell phones and other such hardware devices in their daily activities. This will shorten the exposition and do so without loss of intuitive comprehension. We encourage the reader to draw confidently on his/her intuitive knowledge. It will generally lead to a correct reading of our explanations.

Like many works of genius, the Turing machine concept is deceptively simple but not simple-minded. Over the years after 1936 there have been many variations of the description of a Turing machine but the description we now give has become fairly common. The following description is for the version of the machine propounded by Turing in his 1936 paper. This version, or *model* as it is sometimes called, can be regarded as the original Turing machine. Since then there have been other models (versions) which have equal computing power as the original but are useful in formulating various computation problems. We mention two other models which we shall define at the end of this chapter: the *multi-tape* Turing machine and the *nondeterministic* Turing machine.

A *Turing machine* can be pictured as a device consisting of two parts, a *linear tape and a control unit, C say* (An actual picture is given in Chap. 12). C can activate the reading or writing of characters on the tape. One can visualize an ordinary tape reader-writer device such as those known to Turing in his work on cryptography. The characters scanned can be ordinary alphabetical letters A-Z, for example, or numerical digits 0–9, or useful typographical symbols like *. The tape is subdivided into squares along its length which is specified to be *potentially infinite*, which means that the tape is finite in length but automatically extendible without bound to provide more squares as needed (Think of splicing two tapes together when necessary to obtain more tape space. Of course, since there is only a finite amount of tape in the universe, the potential infinity property is an idealization but it poses no conceptual obstacles to the Turing machine concept. It is a needed property, since real computations may produce arbitrarily long *strings* (sequences) of characters).

C has control of a *read-write head*. For short we shall call this the *head* of C. During a machine computation, which takes place in discrete time steps, C can cause the head to read or write exactly one character per square. We shall include a *blank* character B to represent an empty square. At any instant, the read-write head of C is positioned over one square, the *scanned square,* and its contents, the *scanned character*. The head of C can move left or right from there one square at a time. Alternatively, a physical device might allow the tape to move through C in this manner. How the head moves relative to the tape and how it reads or writes characters is determined by the scanned character and by the *current state* of C, which is an *internal state* (or *machine configuration* of C as Turing called it) at a particular time in the course of its operation during a computation. Turing is rather vague about the concept of *machine configuration*. This is understandable from our earlier remarks about the 1936 milieu which did not provide many concrete examples of computers or other physically active devices and provided only an imprecise notion of the current *state* of a device. Today, we have many experiences with electronic devices and can interpret the concept of current internal *state* as consisting of the collection of current physical states of the active electronic components in a device, such as the conducting or non-conducting states of a transistor (See Chap. 5, Appendix). The main Turing hypothesis about machine configurations that concerns us is that a particular C can have only a finite number of possible states. Turing argued for this property and against an infinite number of possible control states on physical grounds, even speculating about the finiteness of the number of states of a brain. The result is that C is what is now called a *finite-state Control* or automaton. Its finiteness obviously imposes a major constraint on how a Turing machine might be built, that is, it is a *digital* device having a discrete time dynamics of state changes rather than an *analog* device having a continuum time dynamics and infinitely many states. It turns out that the finiteness of C does not limit the generality of Turing computations. Although any particular C is finite, there is no overall bound imposed on the number of states that can be possessed by the control units in all Turing machines. This permits the existence of Turing machines (and computation procedures) of arbitrary size and *complexity* (More on *complexity* in Chaps. 12 and 13).

Turing gave enough examples of his machines to be convincing that his concept was sufficiently powerful to include all known computations and all conceivable ones. We shall restrict our exposition here to one illustrative example, a machine, M, that computes the infinite binary sequence 101010.... Incidentally, we note that Turing was among the first to realize that computers should work with the binary notation of 0s and 1s, rather than with the decimal notation for numbers.

Now, to describe the operation of a machine like M, or any other, we shall introduce the simple notion of a *machine instruction,* I. I consists of a *quadruple* of the form q x O q#, where q denotes the current state of C, x denotes the scanned character (in this example 0 or 1 or the blank character B), O denotes an operation to be executed (Write 0, Write 1, Write B, L for shift left one square or R shift right one square) and finally q# denotes the new state of C after the operation O is completed. M executes such instructions to carry out a computation as a sequence of operations and state transitions. In the Hardware Chap. 5, we shall treat modern computers and it will be evident that Turing's model of machine execution applies to them. As described above, Turing's model embraces six key features as follows: (1) it employs a finite-state Control; (2) a finite number of "internal" (Control) states; (3) a data tape t as "memory" which combined with the control state q constitutes the *total machine state* (q, t); and (4) a *state-transition* function which depends on: (5) some data part of t in the current total state (i.e. here it is simply the scanned character x) and on: (6) the current control state q to cause the execution of the operation of the *current step* of the computation and the transition to the next state q#. These six enumerated elements of machine execution are found in all modern computers.

Furthermore, to describe how M executes its computation, it suffices to give a table, or list, M(I), of all its machine instructions. M(I) is called a *program* for M. Here is the specific program for the M in this example:

M(I) : q(0)B Write 1 q(1); q(1) 1 Rq(2); q(2) B Write 0 q(3); q(3) 0 Rq(0).

M(I) is available to C, possibly on tape t as a *stored program,* in a format that can be read by the Control C. It is assumed that M starts with a blank tape and C in an initial state q(0) as current state. Thus, the initial scanned character is B. The Control finds that the first instruction in M(I) applies and causes M to Write 1 in the scanned square and transit to state q(1). The new scanned character is 1 and the new current state is q(1), so that the second instruction applies. To execute it the control C causes a right shift and transits to state q(2). In state q(2) with scanned character B, the third instruction applies and C causes a 0 to be written and transits to state q(3). The new scanned character is 0 and the fourth instruction applies causing a right shift and a transit back to state q(0). The scanned character is again B and the first instruction applies again. The *loop* of four instruction executions is repeated an infinite number of times, generating the pair 10 each time.

In modern terms, the list M(I) of four instructions is a *program* for machine M. Each instruction causes a *step* of the computation of M to be executed. From a programming viewpoint, in each instruction, the first state serves as a label or *address* of that instruction and the second state is the label or address of the next instruction to be executed. This is a software interpretation of the *control state* concept. It could be implemented in a Hardware version of control C by an *instruction counter* which counts numerically modulo 4 from 0 to 4 (= 0 modulo 4). Other parts of a Hardware version of C would include the read-write head and logical circuits (Chap. 5, Appendix) to combine the scanned character representation with the instruction counter and thereby generate control signals to shift the read-write head or write characters as specified by the instructions.

The Universal Turing Machine. A Stored-Program Computer

As a mathematician, Turing preferred to specify his machine examples as programs, that is, as Software, rather than as Hardware. One important consequence of this method was that Turing soon recognized that the programs could be converted to numerical representations in binary code, with some special characters for punctuation. He called these program codes *standard descriptions*. It was then a natural, yet exciting, step to realize further that a standard description of a machine M could be viewed as data to be written on the tape of another machine, U. He realized that U could be programmed to decode the standard description of M back into machine instructions which U could then execute itself. We would say today that U could *simulate* the operation of M and carry out the computation which M was designed to perform. Turing showed in some detail how this simulation was to be done. The result was the creation of a *universal Turing machine* U. Perhaps the reader can sketch the various pieces of a program for U, as Turing did and as we shall do later. These pieces were programs involving variables. Executable instantiations of them were invoked by statements substituting data for the variables. Today these program fragments are called *subroutines*.

The universal Turing machine was in fact a *stored-program computer*, since any Turing machine program, such as M(I) above, could be stored on the tape of U along with any data needed by M and U could then execute the procedure defined by M. As noted elsewhere in this book, the stored-program idea is the very foundation of modern digital computing and the key to computer design. It is what unifies the Hardware (physical computer) and Software (programming) sides of Computer Science. Credit for the invention of the stored-program idea is often accorded to von Neumann. However, von Neumann had read Turing's 1936 paper and discussed it with him on several occasions. It is not hard to surmise that the brilliant restless mind of von Neumann very likely absorbed the rather ingenious workings of a universal Turing machine and transformed it into the design of a stored-program computer. The now famous report by von Neumann, H.H. Goldstine, and A. W. Burks disseminated the stored-program idea widely (See Chap. 5 on Hardware).

A Program for a Universal Turing Machine

It is rather amazing to read in Turing's 1936 paper a detailed design for a universal machine U. He was mainly interested in the rather abstract and portentous Decision Problem. Yet we find him doing laborious computer programming of a universal machine U. Indeed, it is tedious and laborious to read his program for U, as it often is to read someone else's computer program. To simplify this task we shall not duplicate Turing's program. Instead we shall outline a program for a machine like U but hopefully easier to read.

Recall that a *program* for any Turing machine M is a list M(I) of instructions I which are quadruples of the form qxOq#. Turing probably tired of writing sublists of such instructions in what were clearly repetitive patterns that differed from each other only in certain scanned characters x or internal states q. He created the notion of a *subroutine,* which is a subprogram*,* say P(a, б, г, . . .), of instructions containing variables a,б, г, . . ., referring to tape characters or control states or operations like L, R, and Write. P itself may have a result or, as we say *return* a value, which is a character or a control state. A subroutine like this specifies a family of programs, each member of the family being obtained by substituting values of the correct *type* for the variables. A member of the family can be caused to execute at some point in the *main program* for U by writing at that point a *subroutine call* statement which names the subroutine and specifies values for its variables. The execution returns a value that can be used in the main program. This is basically the way modern subroutines work in current programming languages (See Chap. 4). Here now is an example of a subroutine from Turing's 1936 paper.

Let P(qx, qy, z) be a subroutine which has as its value a control state. Starting at this control state, the Turing machine U will execute a *procedure* which finds the character z which is farthest to the left on the written portion of U's tape. We assume that the left end tape square has been marked initially with a unique character e. The right end of the tape has two or more successive blank squares marked by B's. Subroutine P(qx, qy, z) scans left to e and then right. If it finds a z then it ends in state qx. If no z is found before the repeated blanks B, then it ends in state qy. Here is a program for P(qx, qy, z). It calls two other subroutines P1 and P2 which have control states as their values. The abbreviation Not z refers to all non-blank characters other than z.

P(qx, qy, z)	e	L	P1(qx, qy, z)
	Not e	L	P(qx, qy, z)

P1(qx, qy, z)	z		qx
	Not z	R	P1(qx, qy, z)
	B	R	P2(qx, qy, z)

P2(qx, qy, z)	z		qx
	Not z	R	P1(qx, qy, z)
	B	R	qy

Using such subroutines, we can write a program for a universal machine U. The program is *a main program* having two parts, Subprogram 1(Sub 1) and Subprogram 2 (Sub 2), The main program executes the computation specified by a standard description M(I), a *stored program*, of a machine M. M(I) is written on U's tape as a list of basic quadruples I each I being separated from the next by a semicolon or other suitable punctuation. M(I) is followed by an *input* data tape t of M. Turing uses even-numbered squares for M(I) and t.

Subroutines will cause U to scan its even tape squares to find the quadruples I in M (I) and the data t. The odd squares are used by U to do its computation steps which simulate those of M. The first instruction I is written at the top (left end) of the list M(I).

Subprogram 1 (Sub1). This is essentially a program which sets up the tools for a scan for instructions I of M(I) using an *instruction counter register* CI made up, say, of odd squares of U marked off by special punctuation symbols. Counter CI is initialized with the initial control state of M. Turing uses a coding scheme like qss...s with n + 1 s characters to denote the control states q(n) of M. Similarly, the code xss...s with n + 1 s characters denotes the characters x(n) in M's tape t. These codewords are stored in even squares of U's tape. Using subroutines like P(qx, qy, z) illustrated above, program Sub1 finds the leftmost q character and extracts the codeword qs for the label of the initial instruction to be executed in M(I). It writes qs in CI. Similarly, Sub1 extracts the codeword xss...s for the initial scanned character on M's tape t. It stores xss...s in a *Scan Register SC* consisting of marked odd squares. It also stores in SC the number locating the scanned square of t (say by counting from the left end of t). The pair [CI, SC] together constitute the *current state* of M. Sub1 then transfers control of U to Subprogram2.

Subrpogram2 (Sub2). The instructions in Sub 2 compute successive *current states* of M and apply the relevant instruction of M(I) to each new current state. This procedure is done in two steps which form a *loop of two steps* (i.e. a cycle of repeated executions) as follows:

Step 1: Scan the list of instructions I in M(I) in the even squares of U for those having the label qss...s in the instruction counter register CI. Examine the character part of each such I and compare it to xss...s in SC. Suppose instruction I* "matches" SC in its character part. This is the instruction to be executed next.

Step 2: Let OP be the operation part of I*. This operation (R, L or Write y) is performed on tape t in the even squares of U. R and L simply increment or decrement the integer count in CI. Write y changes the scanned character in SC to be y. Then the next control state, q# say, is extracted from I* and q# is stored in CI. Finally, Step 2 transfers control of U back to Step 1.

Of course, the loop will end if and when no matching next instruction I* is found.

Other Models of Turing Machines

It is convenient to devise other versions of computers that can be considered to be models of Turing machines. As announced earlier, there are two models that have received special attention in the literature on computation: the *multi-tape Turing machine* and the *nondeterministic* Turing Machine.

A *multi-tape machine*, as the name indicates, has a finite number, m, of different tapes that can be used as its memory (See Chap. 12 for a picture). Each tape can be manipulated by the control unit C just as a single tape is manipulated in the original model. Instructions are provided with operation parts such as "Write x on tape k" where k is an integer $\leq$ m. It is fairly obvious that any computation carried out on an m-tape machine can be simulated by a machine having only one tape (For example, just divide the single tape into squares modulo m by means of special "labeling" squares with integers modulo m).

The second model, the non-deterministic machine, is a bit more sophisticated but we shall present it in a rather elementary way. It uses instructions of the form

$$Q_0 \times \text{Op} \ \{Q_1, \ldots, Q_k\}$$

which specify that when in state Q_0 and reading character x on its single tape the machine performs the operation Op and then transits to any one of the k states Q_i in the specified subset $\{Q_1, \ldots, Q_k\}$. The choice of Q_i is an arbitrary process. Thus the *next* state is not explicitly determined. There is *nondeterminism* in the sequence of control states produced in a computation.This can be depicted by drawing a *tree graph* to represent all of the *state dynamics* of a computation. The *tree graph* (see Chap. 7, Appendix on graph theory) would have a root node containing Q_0 say, as part of the state, where Q_0 is the initial control state as in the above instruction and nodes as possible *successor* nodes labeled $Q_1, \ldots, Q_k$. A computation *step* would be represented by marking a designated edge connecting Q_0 to one of the k successor nodes The particular successor node choice, Q_j say, is arbitrary. The next step would proceed from node Q_j by applying an instruction having Q_j as its current state, again making an arbitrary choice of next control state for the next *level* of the tree. The resulting computation would be represented by a *path* of such arbitrarily chosen connected marked edges.

In the Complexity Theory Chap. 12, such nondeterministic Turing machines are used to solve a computational problem by exhibiting a path having a polynomial number of steps, p(n) where n is the length of the "input" to the problem, and it may not be known whether the problem is solvable by a deterministic machines in a polynomial number of steps. This situation is made possible, and apparent in the examples considered, by an algorithm which takes advantage of the multiple choices available at each state node of the computation tree of a nondeterministic machine, and the resulting multiple paths of possible computations among which there is clearly one of polynomial length.

Note that a nondeterminstic machine's computation tree, which can be infinite in depth, can always be simulated by a standard model deterministic machine doing a breadth-first scan of the tree. But this scan may require an exponential number of steps.

Unsolvable Computational Problems; The Church-Turing Thesis

The reader should read Appendix 1 before proceeding with this part of Chap. 3.

The *Church-Turing thesis* is the hypothesis that Turing machines (or equivalently, lambda calculus effective procedures) provide a sufficiently general definition of computation procedures, that is, any conceivable procedure for a computation can be programmed to be executed by a Turing machine. This seems to imply that Turing machines can solve any computational problem. However, Turing realized that there are problems that cannot be solved by his machines. Remember that his paper was written in 1936, after Godel had already shown that there are undecidable well-formed formulas (wff's) that cannot be proved in first-order logic. A formal proof in first-order logic of a wff F is a computation consisting of a sequence of wff's which are axioms or obtained from previous wff's by the rules of inference and ending with F (See Appendix 1). It clearly can be done by a Turing machine. So if F is a Godel undecidable formula, then no Turing machine that implements first-order inference rules can prove F (See Appendix 1). So there are limits to what Turing machines (or Church effective procedures) can do (Hence the term "thesis" connoting a hypothesis that cannot be proven). Despite the simplicity of the Turing machine concept, it gives rise to various degrees of complexity regarding solvable problems (See Chap. 12 on Complexity).

By analogy with well-known cardinality results about real numbers expounded by Cantor, as for example the result that the real numbers in the interval [0, 1] (represented as infinite binary sequences) are not enumerable, Turing considered the problem of enumerating (i.e. listing in a sequence) all *computable numbers*, i.e. enumerating all the computable infinite binary sequences representing real numbers in the interval [0, 1]. A *computable number* in [0, 1] is an infinite sequence of 0s and 1s that is computed by a Turing machine M. By virtue of their cardinality, not all real numbers are computable since all the Turing machine programs M(I) are enumerable (See below). By enumerating all Turing machines it would seem that all computable numbers are enumerable. All rational numbers (ratios of integers) are clearly computable by carrying out the division in the ratio. Likewise, many irrational real numbers are obviously computable by well-known algorithms. For example, √2,□ and e are computable.

To study the possible enumerability of all computable numbers, Turing took the table (program) M(I) of instructions that defines how a machine M computes and encoded M(I) into a particular format called a *standard description* (SD). The format of an SD for M was influenced by Godel's technique of assigning simple strings to represent subscripts. Thus, the m-configurations (i.e. states of the control) q(0), q(1), . . .,q(n), . . . are coded by such simple strings as DC. . .C, where there are n + 1 C characters for q(n). Similarly, he assigned strings AC. . .C to represent the characters x (0), x(1), . . ., x(n),. . .that can be printed on M's tape. The instructions in M(I) are separated by semi-colons. The resulting string is an SD. He then assigned numerical values to the letters D, A, C and the semi-colon, thereby assigning a natural number to each SD, called the ND. From some computable enumeration (listing), done by a machine P, of the ND's of those machines which compute infinite sequences, there would be obtained a computable enumeration of the corresponding computable sequences. This would seem possible. However, Turing proved that the set of ND's of the computable sequences (i.e. computable numbers) is not enumerable. To prove this, it might be thought that the classical Cantor diagonalization argument which shows that the real numbers (infinite binary sequences) are not enumerable would also work here. Not quite. Here, one must pay attention to the computability requirement for the enumeration process.

To prove that the computable numbers are not enumerable Turing assumed the contrary and obtained a contradiction as follows.

So suppose that all the ND's of machines which compute computable numbers (infinite sequences of 0s and 1s) can be enumerated, that is, printed in a list L by a Turing machine P. For any integer n consider the nth ND in the list L. Decode it to obtain SD(n). For the corresponding machine M(n) having the standard description SD(n) let the corresponding infinite sequence x_n which it computes have entries x_n (m), where m designates the mth digit in the sequence x_n. Visualize that x_n (m) defines a two-dimensional infinite array with row index n and column index m. Now define a new sequence S(n) by the formula

$$(*) \qquad S(n) = 1 - x_n(n),\ n = 0,\ 1,\ 2, \ldots,$$

using the diagonal entries x_n (n) of the two-dimensional array x_n (m), as Cantor did. The sequence of numbers S(n) is clearly different from all the enumerated sequences x_n (m), being unequal for m = n. To complete Cantor's argument that the list L is therefore not complete as claimed, we must here show that the sequence S(n) as defined in (*) above is Turing-computable. Now, S(n) can indeed be computed by a machine M(S) which first applies the hypothesized machine P to obtain x_n. For each n, M(S) uses P to generate the nth ND, which defines the machine M(n) that computes sequence x_n and then lets M(n) compute the element x_n (n). Finally, M uses formula (*) to compute S(n). In this way, the infinite sequence S is computable by machine M(S) which uses the Turing machines P and M(n). Hence, S must equal some sequence x_K in the supposedly complete list L, which, as we have seen, is impossible and hence a contradiction. This implies that the supposed existence of P is false. In fact, this would imply by the above definition of S(n), that $1 - x_n(n) = x_K(n)$ for all n. In particular, for n = K we would have the contradiction $1 - x_K(K) = x_K(K)$, that is, $1 = 2\, x_K(K)$, making the number 1 even or 0. Therefore, the supposition that a Turing machine P exists to enumerate all the sequences x_n (m) is false.

So Turing machines compute all computable numbers (by definition) but fail to enumerate all computable numbers, as might seem possible by naive reasoning. In fact, Turing goes on to consider a related problem: Find a Turing machine P to decide whether or not a DN is the DN of a machine which computes an infinite binary sequence. Clearly, we can construct a machine which computes only a finite binary sequence, say by arriving at an end (or *halt*) m-configuration q for which there is no instruction with the label q, or by getting into a *loop* of instructions which simply causes the read-write head to oscillate back and forth without printing and never halt (the dreaded *infinite loop* in modern programs). Such simple behaviors can be detected by examining short programs. But the DN may be that of a long complicated program. Can we design a Turing machine P which can decide if any DN defines what Turing called a *circular* machine, one that does not compute an infinite binary sequence (i.e. halts after a finite number of instruction executions or else gets trapped in an infinite loop (or circle) of executions). A non-circular machine he called *circle-free*. We already know that P cannot exist, since if it did the ND's of circle-free machines, which compute infinite sequences, would be enumerable. In simpler terms of machine behavior, the question is: can a machine P be designed to detect circular machines? P would be called a *debugging diagnostic* program in modern computer programming. There can be no such Turing machine P, despite the vast powers of the class of all Turing machines.

Appendix 1

Symbolic Logic, Computer Science and the Godel Incompleteness Theorem

The logicians B. Russell and A. Whitehead (R&W) in their monumental 1910 book Principia Mathematica attempted to derive all of mathematics within a *formal system* of symbolic logic following an earlier similar attempt by the logician G. Frege. In a related approach, the *formalist* school of mathematicians, led by David Hilbert, also in the early 1900s, advocated a process for the mechanization, by formalization within a *formal system* of logic, of all of mathematics. According to this formalist process, all of mathematics, (i.e. the theorems), is to be organized as one grand deductive logical system of formulas (called *well-formed formulas* or *wff's* for short) obeying precise syntactic structural rules and the true statements of mathematics (the *theorems*) are to be represented symbolically by wff's which are *provable* (mechanically derivable) in the logical system by *formal proofs* from appropriate *axioms*.

In a *formal proof* one proceeds step-by-step in an almost thought-free manner solely by applying the purely formal rules of the deductive logic system, a process which could seemingly be done by a computing machine if one existed. Indeed, some modern computer programmers have explored this possibility of proving some theorems by machine methods. The formalists' doctrine, surprisingly proposed by a prominent mathematician like Hilbert who had done many important *informal* mathematical proofs in the usual intuitive natural language style, aimed to reduce all mathematical theorem-proving to purely formal symbol-manipulation computations with no intuitive understanding required of the interpreted meanings of the statements in the various steps of a proof. This proposal may have been suggested by the well-known quasi-formal proofs in abstract geometry presented in Euclid's Elements (See Chap. 2 and section "The Decision Problem of Formalist Mathematics"). As remarked skeptically by other mathematicians, notably the eminent pure mathematician G. H. Hardy, such a program, if successful, would reduce all mathematical theorem-proving to a mechanical thought-free process of symbol-manipulation, one that could in principle be carried out by some kind of computing machine. Indeed, as we have said, modern computer scientists have tried to develop theorem-proving computer programs. Hopefully, by reading this Appendix 1 they may learn their limitations. Of course, even in the formalist program some human thought would be required to set up appropriate axioms for each mathematical theory. This seemed quite possible since it had already been done for the theory of natural numbers N by the mathematician Peano (See below). Other theories using real numbers could presumably be based on N. So the formalist program seemed reasonable and it began with N.

In this Appendix 1, we shall describe how the then (1931) young mathematician Kurt Godel upset the formalist program by proving that its grandiose goal is not achievable even for N. He did this by displaying a wff which clearly is interpreted as a true mathematical statement about N (by an obvious interpretation in N) and yet is demonstrably not formally provable in the formalists' well-accepted logical system. We wish to explain the essence of Godel's result and its significance for Computer Science without getting distracted by the syntactic details of the encompassing logical system. Yet, we must describe the main features of that system to explain Godel's result. Hence, we shall present the main features but omit many syntax details about the encompassing formal logical system, such details being either obvious or well-known to those readers who are familiar with at least one computer programming language, which in several ways can be viewed as a formal logical system. For those who need more detailed explanations we refer them to such standard references as D.Hilbert–W. Ackermann (H&A) Principles of Mathematical Logic. Translated, Chelsea 1950. We shall refer to the formal system in H&A as HA. Also see D. Hilbert and P. Bernays Foundations of Mathematics, Springer, 1931. This appendix omits many of the details about the HA formal system (e.g. the complete syntax of wff's), but we shall at least explain the motivation for how the formal logical system was constructed by H&A (following R&W) and how its deduction (*inference*) rules were intended to be used by the formalists as a "thought-free" machine-like proof mechanism.

Propositional Logic and Boolean Algebra

At the outset, it is important for us to recognize that although the wff's in a symbolic-logic system on the one hand are treated purely as *formal* symbolic expressions having no meanings, on the other hand they are *interpretable* intuitively as meaningful statements that occur in natural-language treatises involving concepts of informal logic and mathematics. This is somewhat analogous to viewing a computer programming language, on the one hand, as a medium in which to simply write symbolic formulas (e.g. wff's like assignment statements or Boolean expressions) as prescribed by a precise syntax but not giving much thought to their possible meanings while, on the other hand, interpreting these wff's as natural-language statements about computations in some external domain like N. Furthermore, these interpreted statements, about N say, have an ordinary *truth-value* (True or False), which, as we shall see, is transferred back to their symbolic wff's by a computable procedure.

The formal logic system HA has certain logic *axioms*, which are wff's formalizing statements which are intuitively true under any natural interpretation. The axioms can be used as steps in formal proofs. As we have remarked elsewhere, Euclidean geometry serves as a prototype example of such a formal deductive system in which the meanings (i.e.in that case, geometric interpretations) of the statements are not involved in the proofs. In fact, there are no syntactically precise wff's in Euclid's Elements, However, the points and lines mentioned in quasi-formal statements in proofs are to have no geometric meanings and are to be treated as abstract objects having only such properties as are conferred on them by Euclid's axioms; e.g. any two *points* determine a *lin*e. Euclid's proofs of theorems proceed from the axioms by applying computational deductive rules of logic. For a major example of a computational deductive rule used in a formal proof in any logical system we refer to the well-known *modus ponens* rule below.

For our purpose here, we do not need to give the complete syntax of well-formed formulas (wff's) in the logic system. A few examples will convey the main features of wff's. The interested reader can refer to Hilbert& Ackerman for a full account. Russell & Whitehead (R&W) and Hilbert & Ackerman (H&A) set up their formal logic systems starting with formulas called *propositions*. They begin with *propositional variables* p, q and r etc. which are meant to denote arbitrary natural language declarative sentences which are either true or false, for example "Socrates was a man." or in a modern computer context a statement like "switch 1 is closed." Syntactically, variables like p, q and r serve as *atomic propositions*, that is, they are the basic building blocks of propositional wff's. Analyzing how mathematicians construct informal intuitive proofs, R&W and H&A postulate that they mainly use the *logical connectives* AND (denoted by &), OR (denoted by v) and NOT (denoted by ¬) to construct *compound propositions* represented by such wff's as p&q, p v q and ¬p. They also use the wff p → q to denote an *implication* which formalizes the statement "p implies q" or equivalently "if p, then q." More complicated wff's are constructed without limit by substituting wff's for (all occurrences of) a propositional variable p, q, r etc. in previously formed compound wff's or by repeated use of the logical connectives, for example like (p → q) → r. To compute the truth-value of a proposition when given truth values for the interpreted statements assigned to the propositional variables in the wff it is only necessary to know how the connectives determine truth-values. This is easily given by the natural *truth-tables* for the connectives as follows (with truth = T and falsity = F):.

p	q	p → q	¬p v q
T	T	T	T
T	F	F	F
F	T	T	T
F	F	T	T

p	q	p v q	p & q	¬ p
T	T	T	T	F
T	F	T	F	F
F	T	T	F	T
F	F	F	F	T

Clearly, as shown, a *conjunction* formula p & q is true exactly when the variables p and q are both interpreted as true. A *disjunction* formula p v q (denoting the *inclusive* OR) is true if either p or q is true or both are true. This agrees with the truth values of natural language intuitive statements formed using these connectives. In passing, we observe that p → q and ¬ p v q have the same truth values.

Using these truth tables, for any propositional wff, f say, containing precisely the n propositional variables p(1),. . ., p(n) we can compute the truth value of f from an assignment of truth values to the variables p(1),. . ., p(n). Indeed, f defines a *truth function* of these n truth values. In the computation of the function's truth value f(p(1),. . ., p(n)) we can regard the connectives as basic truth functions, or algebraic *operations* defined by their truth tables. Thus, in propositional logic, the truth of a wff under any interpretation of the atomic variables as natural language statements can be computed solely from the truth values of these interpreted statements. We do not need to consider the meanings of the statements. This situation arises in the design of computer switching circuits where there are n components, like transistors for example, which are either conducting ("on") or non-conducting ("off"). We choose n proposition variables x(j), j =1,. . .,n to denote the statements "Component j is on." and assign truth value T if it is "on" and F if it is "off". It is then required to design a circuit which computes a prescribed truth function f(x(1),. . ., x(n)) which denotes the statement "The circuit output is on.", which is true when the output line is "on" and false otherwise. The design of f can be done using algebraic methods known as *Boolean algebra* and hardware for f can be built using well-known circuits called *gates* for AND, OR and NOT.

In fact, about 50 years before R&W and H&A, in 1854, the English mathematician George Boole published his book "An Investigation of the Laws of Thought" (republished by Dover Pub. Co., N.Y. 1958) in which he proposed an abstract algebra of propositions in which the connectives behave like algebraic operators. We now call this *Boolean algebra* and it is used in Hardware design (see Chap. 5, Appendix) and also is incorporated in modern programming languages along with the usual arithmetic operators + (addition), • (multiplication) and – (negation) (See below).

Boole also considered aspects of human logic reasoning dealing with subsets of some given set, for example subsets of the set of integers N. In this type of reasoning, the *union* of two subsets X and Y, denoted by $X \bigcup Y$ and the *intersection* $X \bigcap Y$ arose in a natural way, as did the complement X'. The union $X\bigcup Y$ arose for a proposition p V q where p is true for X and q for Y. Then p V q is true for elements in either X or Y or both, that is, in $X\bigcup Y$. Likewise, p&y is true for $X\bigcap Y$. Boole used 1 for true and 0 for false. Thus p and q were assigned one of the *Boolean values 0 or 1.* p and q were *Boolean variables* representing propositions that were either true or false in a proof in a propositional logic system. For a compound propositional wff W involving n proposition variables, W would have different truth values for different truth values assigned to the n variables. An arbitrary wff W in propositional logic defines a *Boolean function* when we restrict the values of the propositional variables in W to be either T or F, as when we are only interested in the truth values of W (rather than its meaning).

Boolean Algebra and Circuits

Boole recognized that the computation of truth values of an arbitrary wff can be done within an abstract algebraic setting involving two *binary* operations, + and •, corresponding to the OR and AND logic connectives v and & respectively and a *unary* operation ′ corresponding to NOT (We assume that p → q has been replaced by ¬ p v q). In modern mathematical terms, Boole's algebraic setting is called a *Boolean algebra*. It consists of an abstract set B with *binary operations* + and • and a *unary operation* satisfying the following axioms:

1. For x and y in B, the elements x + y and x • y and x′ are in B;
2. There are elements 0 and 1 in B such that x + 0 = x and x • 1 = x;
3. x + y = y + x and x • y = y • x;
4. x • (y + z) = x • y + x • z and x + y • z = (x + y) • (x + z);
5. For any x in B there is an element x′ such that x + x′ = 1 and x • x′ = 0.

Axiom 1 is really not needed, since it follows from the definition of an *operation* in an algebra.

An example of a Boolean algebra is the set B = {0, 1} with the Boolean + and • operations defined as the usual arithmetic operations except that 1 + 1 = 1. Also, 0′ = 1 and 1′ = 0.

Another example is the set all subsets of a given set X. with the + operation being union and the • being intersection and x′ being the complement of x. Here 1 is X and 0 is the empty set.

We have already mentioned the application of Boolean algebra to computer switching circuit design involving n "input" components x(1),. . ., x(n) which, like transistors, (Chap. 5, Appendix) can be in either of two states "on" (= 1) or "off" (= 0). The components are usually to be connected through OR and AND and NOT *gate* elements so that there is a single circuit output f(x(1),. . .,x(n)) which is either on (= 1) or off (= 0) as required for the specified circuit operation (See the Hardware Chap. 5). Thus the circuit computes a specified *Boolean function* f. In principle, the definition of f can be given by a table of its values for the 2^n possible *Boolean vector* values (x(1),. . .,x(n)). Each row of the table will list the values of the x(j) variables (0 or 1) and an extra column for the specified value of f. We consider only those rows in which f has the value 1. A Boolean formula for f can be written as a *disjunctive normal form* consisting of sums (disjunctions) of products (conjunctions) **y(1)** • **y(2)** … • **y(n)** of n input variables y(j), where y (j) = x(j) if x(j) = 1 in that row and y(j) = x(j)′ if x(j) = 0. This conjunction will have the value 1for the value of (x(1),. . .,x(n)) in that row. The sum of these conjunctions will have the value 1 exactly for those vector inputs where it is required and 0 otherwise. This combination of OR (sum), AND (product) and NOT gates will compute f. However, it is probably not the minimal circuit to compute f. Algebraic simplification according to the Boolean axioms may be needed. Also it may be better to use other types of gates such as NAND gates (See the Hardware Chap. 5).

Propositonal Logic and First-Order Predicate Logic

To carry out formal logical proofs with formal symbolic expressions, called wff's, representing natural language statements, H&A first formulated the system called *propositional* logic for the propositional wff's described above. These wff's are interpretable as natural language statements which are either true or false. For doing formal proofs with propositions, H&A chose certain wff's as *axioms* to serve as initial wff's in a proof. The following four axioms were chosen:

Let U, V, W be wff's which can be interpreted so as to have truth values. For example U, V and W can be propositional variables. The axioms are:

1. $W \vee W \rightarrow W$;
2. $W \rightarrow W \vee V$;
3. $W \vee V \rightarrow V \vee W$;
4. $(V \rightarrow W) \rightarrow (U \vee V) \rightarrow (U \vee W)$.

Certainly, the first and third formulas are intuitively true for any interpretation of W and V as natural language statements. Therefore, they are said to be *valid*. Their truth value, TRUE, for any assignment of truth values to V and W can be established by truth tables as in the above section on Boolean algebra. The same holds for axioms 2 and 4. As written here, the axioms are really axiom *schemas* since U, V, W can be any wff's.

A *proof* is a sequence of wff's starting with an axiom. At each step in a proof the next wff in the sequence (i.e. the next wff proved) is an axiom or is determined by either of two *rules of inference* applied to wff's already in the sequence. The rules are

1. The *substitution rule,* which allows any wff to be proved by substituting a wff for all occurrences of an atomic variable in a wff already proved
2. *modus ponens*, which allows a wff Q to be *proved* if a wff P and the wff $P \rightarrow Q$ have already been proved in the current or previous steps of a proof.

It is clear by induction on the length of a proof that the rules of inference at each new proof step *preserve the truth* of earlier steps. The axioms are true and modus ponens yields a true wff Q, since P and $P \rightarrow Q$ are true by the induction hypothesis. Therefore, starting a proof with a purely logical axiom we can only prove wff's which are propositions that are true for all possible interpretations. This was not the formalist program. They proposed to do formal proofs of wff's which are interpreted as mathematical statements, say about natural numbers for starters. To prove wff's which are interpreted as mathematical statements in a formal system, the formal system must provide symbols and axioms of a mathematical type. In modern computer programming terms, it is necessary to have variables and operators of type "integer" if we wish to have formulas that refer to integers. In the formalist program, it is reasonable to begin with the mathematics of the natural numbers, N, and adjoin to the logic system symbols to denote numbers and wff's to denote axioms about N, thereby extending the logic system to a formal system F(N) for N.

Before doing this, it is necessary to further extend the propositional logic system to take into account general mathematical statements which contain variables that can take on values in a mathematical domain like N. This leads to an immediate extension of propositional logic called the *first-order predicate logic*.

By analyzing how logic is used in mathematical proofs, for example in Euclidean geometry and number theory, R&W and H&A were able to extend the propositional logic system to a more general logic system which *formalizes* the intuitive logic methods for mathematical domains like the non-negative integers N. As with the propositional logic system defined above, the first step of this extension process was the representation of mathematical language statements by abstract *well-formed formulas* (wff's) according to a precise syntax. In mathematics, typical statements contain *individual variables* like x, y, z which range over the particular domain being investigated, for example the integers N. Statement have a subject-predicate structure like "x is P" where P is a predicate like "a prime number"; i.e. "x is a prime number." To formalize such statements H&A adjoin to the propositional logic new symbols called *predicate variables* say P, Q, R etc. which denote abstract predicates and they then define *predicate formulas* (wff's) like P(x) to symbolize the statement "x is P". This functional notation indicates that P(x) denotes a propositional function, that is, for each value, c, substituted for the individual variable x the formula P(c) symbolizes the proposition "c is P". Thus the interpretation of a wff symbol like P(x) is an abstract unary relation (i.e. a subset) of some domain like N. It has no truth value. More generally, a wff like P(x, y) denotes a binary relation on some set like N, as for example is defined by the statement "x = y + 1" and so on for n-ary relations for each number n. These statements do not have truth values. These are the *atomic predicate wff's*. As with atomic propositions, the atomic predicate formulas are building blocks which can be combined by the logical *connectives* into abstrac*t compound predicate* wff's. Thus, P(x) & Q(y) and P(x) v Q(y) are compound predicate wff's. In the formal system F(N) for N we also have specific atomic predicate wff's like "x = y + 1". The individual variables x and y in such predicate wff's are said to be *free* variables. They can be assigned any value in a specified domain like N. Symbolically to represent such an assignment of a value to a variable x it is permissible to substitute a constant symbol, say c, for all occurrences of a free variable x in a wff, thereby creating a predicate wff like P(c), which does have a truth value when interpreted in F(N), as for example "c is a prime number.".

Now let W(x) be a predicate wff containing the free variable x. Besides substituting a constant c for the free variable x in W(x) so as to obtain a truth value for the resulting wff W(c) it is also possible to obtain truth values by applying *quantifiers* to *bind* x, as is done in ordinary mathematics exposition. A free individual variable x in a predicate wff W(x) can be *bound* by being *quantified* by prefixing the *existential quantifier* ∃x, denoting "there exists x" or the *universal quantifier* ∀x denoting "for all x". Then if there are no other free variables in W(x) the wff ∃xW(x) is a wff with no free variables and can be assigned a truth value. For example, we could have ∃x (x − 2 = 0). Such predicate wff's with no free variables obviously do have truth values. We repeat that if x is not bound in W (x), x is called a *free* variable and its occurrences can be replaced in the usual way by *substituting* for them a *value,* that is, a symbol c denoting a constant in some interpreted domain, for example a number in N. But the predicate variables P, Q, R etc. cannot be quantified. This is the meaning of the adjective *first-order.* This completes the syntax of predicate wff's. The resulting symbolic logic system with two more axioms given below is called the *first-order predicate logic*, in recognition of the property that predicate variables cannot be quantified.

For deductive proofs in first-order predicate logic H&A adjoin to the propositional axioms the following two predicate wff's as axioms which specify how quantifiers can be used in proofs.

Additional Axioms of First-Order Predicate Logic

Let W(x) be a predicate wff containing the free variable x and let t be a term which denotes an element in some domain like N. Then the following are axioms:

$$\forall x(W(x)) \rightarrow W(t);$$

$$W(t) \rightarrow \exists x P(x).$$

Note: By a *term* t is meant a constant c or some combination of constants and operations (like 2 +3) which denotes an element in some domain like N. The first axiom allows the universal quantifier to be eliminated in a proof. Thus if ∀x(W(x)) has been established in a proof step, then applying this axiom and modus ponens allows the derivation of W(t) in the proof. Similarly, if W(t) has been proven, then the second axiom and modus ponens allows the derivation of ∃xP(x). These are natural mathematical proof steps.

For predicate logic the four axioms of propositional logic still hold but with the wff symbols U, V, W now denoting predicate wff's which have truth values. Further the rules of inference of propositional logic still hold but again for predicate wff's as well as propositional wff's.

The same two rules of inference are adopted to derive (i.e. prove) other predicate wff's in a formal proof. These are

1. The *substitution rule,* which allows any predicate wff to be substituted for all occurrences of an atomic variable
2. *modus ponens*, which allows a predicate wff V to be proved if predicate wff's W and W → V have already been proved.

The resulting formal logic system is called the *first-order predicate logic.*

As with propositional wff's it follows easily by truth table analysis using Boolean algebra that any predicate wff provable by logical deduction from wff's which are intuitively true must likewise be true. We shall describe this behavior of truth by saying that *truth is preserved* by the logical proof system and call this property of the system *truth-preservation.* This property of the logic system is sometimes also called *soundness.* It is easy to show that the property holds for modus ponens.. (If W is true, then W → V can only be true if V is true. So modus ponens yields only true V.) Since the axioms are valid (true for any interpretation), so are all provable predicate wff's. This is an important metalogical property of the logical system that we certainly require if the system is to be of any value in proving theorems. It partly justifies the Hilbert formalist program. To completely justify it another metalogical property is required, a converse of truth-preservation, namely, that any wff which is valid (true by any intuitive interpretation) must be provable. If this property held, the logical system would be considered *complete.* Formal proofs of valid wff's would be a mechanical symbol-manipulating computation. In his 1929 doctoral thesis, Godel showed that the first-order predicate logical system is complete. However, valid wff's were not the wff's of interest in the formalist program to mechanize mathematics. The program rather dealt with wff's which are interpretable as true in particular mathematical domains. The first such domain to be formalized was the non-negative integers (natural numbers) N with its usual arithmetic. This was done by adjoining wff's to denote statements about N and adjoining axioms for the natural numbers N to the logical axioms so as to construct a formalized number system F(N). The formalist dogma was that every wff in F(N) which is interpretable in N as a true statement about N is formally provable in F(N). This would make N a domain similar to Euclidean geometry. It would also make F(N) a *complete* formal system for N in that for every true statement S about N there is a wff F(S) in F(N) which can be interpreted as S and which can be formally proved in F(N). In fact, Godel proved that F(N) is not complete, that is, it is *incomplete*, by displaying a wff S which is true in N but its formalization F(S) is not provable in F(N). He further showed that its incompleteness cannot be remedied, say by adjoining more axioms. As we shall see, this result has implications for problems in computation.

In his earlier 1929 doctoral thesis, Godel established that the first-order predicate logic system is complete. From the way that any logical system works, if W is a provable formula, then ¬W is not provable, since W must be true by the truth-preservation property and so ¬ W must be not true, hence not provable. This is also called a form of system *consistenc*y. Godel's incompleteness result assumes that F (N) is consistent.

To actually formalize some branch of mathematics within a symbolic logic system it is necessary to adjoin symbols which denote mathematical objects in that branch, for example numbers in N, and to adjoin axioms which denote properties of these objects. A similar process is used to design a programming language for N. Once the purely logical axioms have been chosen for first-order predicate logic as explained earlier, it is natural to adjoin axioms for the arithmetic of natural numbers, N. One possible set of axioms for N are the well-known Peano axioms which employ the constant 0, the successor function s, addition +, multiplication • and equality =. These axioms are given by the following six purely arithmetic formulas and a seventh formula involving logic:

Peano's Axioms

1. $\forall x \neg(0 = sx)$;
2. $\forall x\, \forall y\, (sx = sy) \rightarrow (x = y)$;
3. $\forall x\; x + 0 = x$;
4. $\forall x\, \forall y\; x + sy = s(x + y)$;
5. $\forall\, x\, \forall y\; x \bullet sy = x \bullet y + x$;
6. $\forall x\; x \bullet 0 = 0$.
7. Let W(x) be a wff interpretable in N and having a free variable x. Then

$$(W(0)\ \&\ \forall x(W(x) \rightarrow W(sx)) \rightarrow \forall x\, W(x).$$

The intuitive interpretations in N of these axioms are quite obvious. For example, axiom 1 can be interpreted as stating that 0 is not the successor of any natural number. The Peano axioms have been found to be sufficient for deriving known properties of N. Three additional axioms are adjoined to represent the usual properties of equality. The result is a formal system for N. Let us continue to use F(N) to denote this formal system within first-order predicate logic with the Peano axioms adjoined.

Axiom 7 is the principle of *mathematical induction* to prove ∀x W(x). As in the axioms of predicate logic it is really an axiom schema since W(x) is an arbitrary predicate wff. Strictly speaking, there is an implied universal quantification of W, making this axiom look like a second-order wff. However, the universal quantifier ∀W is not permissible and is not part of the formal syntax. It enters informally by the interpretation of axiom 7. See the remarks below about the interpretation process.

What Godel did to show the incompleteness of F(N) was to *arithmetize* the well-formed formulas and the deductive proofs that take place in F(N) by using the formal arithmetic provided by F(N) itself. In this process, the formal objects of F(N) (i.e. the well-formed formulas and the proofs) are assigned numerical codings. His paper gives methods for the effective calculations of all the numerical codings. We shall sketch the key ideas. In the following, x, y and z are individual variables that can be assigned numerical values.

The first idea in Godel's methods is to assign formal numerical values (we shall loosely say "Godel numbers") to the formulas in F(N). Exactly how he does this we shall not describe but it is easy to see that it is by a simple effective calculation expressible in F(N), which first assigns numerical values to individual symbols. Then for a well-formed formula, say the symbol string f, he combines the numerical values for the symbols in f into a single number, G(f), called its *Godel number*. G is a computable function, or as Godel does it, a *recursive* function, which is a concept equivalent to Turing computability.

Second, he considers the set of all formulas $Y(\eta)$ in F(N) which contain exactly one free variable η. For simplicity he uses the same variable η. For each such formula he defines its Godel number $G(Y(\eta))$ within F(N) in a way that permits the arithmetic statement "y = Godel number of Y(η)" to be expressed by a formula in F(N). This assigns G(Y(η)) to the variable y. Next, he gives a formula expressing that "Y (z) is the result of substituting z for the occurrences of the free variable in Y(η)".

The final idea is to compute a Godel number G(pfX) for any proof pfX in F(N). A wff in F(N) is represented as a number computed from the sequence of the Godel numbers given to the sequence of symbols in the wff. Similarly, a Godel number of a proof pfX is computed from the sequence of numbers of the wff's in the steps of pfX. This is all done in such a way that the intuitive arithmetic assignment statement "x = G(pfX)" can be expressed by a formula in F(N). These numerical values for the formal objects in F(N) *arithmetize* its formal aspects in a computable way. Thereby, the formalism of F(N) becomes a branch of ordinary arithmetic.

Finally, Godel formulates a complex predicate expression in F(N),

$$\mathrm{Prf}(\mathrm{x}, \mathrm{y}, \mathrm{z}),$$

involving three free variables x, y, z, which has the following three-part interpretation (meaning):

Part (1) y is equal to the Godel number of a formula Y(η) with one free variable;
Part (2) Y(z) is the formula obtained by substituting z for η in that formula Y(η);
Part (3) x equals the Godel number of a proof pfX of Y(z) in F(N).

Part (1) might be a formula such as "y = G(Y(η))", where G(Y(η)) is a Godel number that holds precisely for a wff which contains one free variable,η.

Part (2) gives a Godel number for the syntactic result Y(z) of substituting z for η in the formula Y(η) given by Part (1).

Part (3) obtains the Godel number for Y(z) from part (2), decodes it as wff Y(z) and then computes a Godel number for a proof, pfX, of Y(z) and asserts

$$\mathrm{x} = \mathrm{G}(\mathrm{pfX}).$$

Of course these high-level formulas and interpretations have recourse to various predicates having lower-level interpretations based on F(N) syntax and inference rules such as "pfX is a proof of Y(z)" which will usually involve definitions by induction on the length of a well-formed proof sequence pfX ending with Y(z). In effect, Godel arithmetizes the syntax of formulas in F(N) and the construction of formal proofs in F(N) by defining suitable numerical predicates in F(N) and uses these to formulate the predicate Prf(x, y, z). We assume, as has been checked by others, that Godel's paper defines these numerical predicates correctly. Hence, we assume that Prf(x, y, z) is correctly and effectively constructed and the three-part interpretation of Prf(x, y, z) is as specified above. The rest of Godel's proof can proceed directly as follows (But see the addendum at the end of this appendix).

It may be helpful to observe that Prf(x, y, z) as interpreted above sets up a two-dimensional array of wff's indexed by the integer-valued pairs (y, z) where the row indices y are Godel numbers which designate one-variable predicate wff's in F(N) and the column indices z designates numerical values to be substituted for the variable in these wff's. Since the particular free variable in a one-variable predicate may be any variable, we can choose a convenient one to use in all. In a way, the rest of Godel's proof can be viewed as applying a version of the classical Cantor diagonalization procedure to this array as follows.

Consider any value of row index y and obtain its designated one-variable formula, say Y(η), essentially by inverting the function G. Thus, y = G(Y(η)). Now take the column index z to equal y, (thereby selecting the diagonal elements of the array) and consider the wff Y(y) obtained by substitution of y for η as given in part 2 of the interpretation above. This yields the "diagonal" cases of the predicate Prf(x, y, z) namely, the predicate formula Prf(x, y, y). These cases will be used to generate an unprovable formula. Godel proves that one special instance of the various wff's Y(y) is not provable in F(N). He considers the special one-variable predicate wff U(y) in F(N) defined as

$$\mathrm{U(y)} : \neg\exists\mathrm{x}\ \mathrm{Prf(x, y, y)}$$

which is interpreted as meaning that there is no proof Godel number, x say, of any wff Y(y) in which y equals the Godel number of Y(η). Taking one final step, Godel then computes the Godel number u of the predicate U(y), namely, u = G(U(y)), and constructs the specific formula U(u) given by

$$\mathrm{U(u)} : \neg\exists\mathrm{x}\ \mathrm{Prf(x, u, u)}.$$

Formula U(u) has no free variables since u is a Godel number. The interpretation of U(u) states, by carefully unwinding the steps in the above interpretations (1),(2), (3) of Prf (x, u, u) that there is no proof Godel number x of U(u). So U(u), interpreted, asserts its own unprovability! As Godel pointed out, this *self-referential* property of U(u) is not new to logic. It was known in traditional logic as some variation of the *liar's paradox* statement "**I am lying**" (Dear Reader: Can you decide if the liar is lying or telling the truth? Is his statement true or false? Either decision will lead to a contradiction).

In fact, the **interpretation** of U(u) is a metatheorem about provability, or rather a lack of it, in F(N). It is the essence of Godel's incompleteness result. The truth of the metatheorem is easy to establish mathematically by a standard, simple, informal proof by contradiction, as Godel did, and as we now do.

Godel's Incompleteness Theorem Let u be the Godel number of formula U(y) above. The formula U(u) given by ¬ ∃x Prf (x, u, u) is true by interpretation but not provable in the formal system of number theory F(N). Likewise, the negation of U (u) is not provable (i.e. U(u) is *undecidable*).

Proof Assume that the formula U(u) is provable in F(N). Then by truth-preservation the interpretation of U(u) is true. But the interpretation of U(u) asserts its own unprovability. Hence, U(u) is not provable. This is a contradiction i.e. the assumption that U(u) is provable implies U(u) is not provable. Therefore, by classical logic, U(u) is not provable. Furthermore, this unprovability is exactly the interpretation of $\neg\exists$x Prf (x, u, u), so that this formula (i.e. U(u)) is true. So F (N) is incomplete.

Finally, consider the negation $\neg$U(u). Since U(u) is true, $\neg$ U(u) is not true. Therefore, $\neg$U(u) is not provable. #

Beyond Computation. Formal Syntax and Informal Interpretation

This theorem has raised questions about differences between the human brain and *machine intelligence*, a subject which interested Turing. E. Nagel and J. Newman in their book *Godel's Proof*, N. Y. University Press, 1958, find in this theorem a rejection of machine intelligence. The physicist Roger Penrose in his 1994 book *Shadows of the Mind* conjectures that human consciousness is beyond computation (as defined and limited by the Church-Turing thesis) and speculates that, by some as yet unrecognized quantum mechanics processes the brain may be able to perform non-discrete algorithmic tasks that exceed the capability of Turing machines, such as establishing the truth of $\neg\exists$x Prf (x, u, u). Ongoing research on quantum computers (Chap. 14) may be trying inadvertently to exploit this possibility.

The Godel theorem on incompleteness of F(N) ended Hilbert's formalist program. It may be thought to cast some doubt on the adequacy of formal deductive proofs as implemented and limited by the Church-Turing thesis type of computations. However, our presentation of the Godel theorem does not force us to such drastic conclusions. As we presented it, the key to the theorem is in drawing a careful distinction between the formal *syntactic* aspects of the logical sys*tem* F(N), which can be programmed into Turing machines, and *the semantics* of F(N), which we have associated with its *interpretation*. The *interpretation* process is not subject to the Church-Turing thesis. It can possibly be treated as a complicated mapping from F(N) wff's onto natural language statements and their subsequent truth analysis using intuitive number theory N. This mapping has not been restricted to Church-Turing computability. To completely define the interpretation mapping would entail some computable constructions which characterize deductive proofs, as Godel has done, and furthermore satisfy the principle of truth-preservation. But more is involved. One of the elusive issues in defining the interpretation mapping is that of specifying the *semantics of truth*, especially where natural language is concerned. Natural language can be ambiguous, making truth analysis inexact. This issue has been studied by linguists and by various schools of logic, especially the *intuitionis*t school of logic, led by L.E.J. Brouwer who raised questions about classical logic in his paper *The untrustworthines of the principles of logic*, 1908, *Tijdschrift voor wijsbegeerte*. For example, the intuitionists do not accept the classical *law of the excluded middle* which posits that any meaningful statement P is either true or false, which means that P OR $\neg$ P is always a true statement. More precisely, for intuitionists P XOR $\neg$P is true, where XOR is the *exclusive or* connective in which there is no middle true position (i.e. P XOR Q is not true if both P and Q are true as in the *inclusive* P OR Q, which is true if both P and Q are true).

In particular, they do not accept this law for a statement P concerning an infinite set, which they regard, as Gauss did, as an incomplete entity. See S.C. Kleene, Introduction to Metamathematics, 1950, van Nostrand, for further discussion of intuitionism. Although fascinating, we shall not consider these issues further. We shall accept classical logic in our study of Computer Science and employ classical logic notions of mathematical truth such as the P OR¬ P law.

Hilbert died in 1943. It is rather ironic and sad to read the inscription on Hilbert's tombstone, taken from an address that he delivered upon his retirement in 1930. He insisted that there are no unsolvable problems in mathematics and science. He said:

> Wir mussen wissen (We must know.)
> Wir werden wussen (We will know.)

It was only days preceding his address, and unknown to him, that the young Kurt Godel at a symposium on the foundations of mathematics announced the Incompleteness Theorem described in this appendix. There are *undecidable* statements of conditions which we can never "know" by proving them.

Addendum. Actually, the preceding explanation of Godel's proof is lengthier than Godel's own explanation of his proof. The latter can be read in the English translation of Godel's paper by Elliott Mendelson in the book "The Undecidable", edited by Martin Davis, Dover Publishing Co, 2004.

Godel sketches the main ideas of his proof at the beginning of his paper before plunging into the detailed constructions of the various wff's involved. We quote some of his original statements with our own added parenthesized remarks. Thus, "For metamathematical considerations it makes no difference which objects one takes as primitive symbols…we decide to use natural numbers…a formula is a finite sequence of natural numbers … and a proof-figure is a finite sequence of finite sequences of natural numbers… the concepts *formula*, *proof-figure* [lines of a proof], *provable formula* are definable within the [formal] system…we obtain an undecidable proposition A for which neither A nor not-A is provable." A formula with exactly one free variable and of the type of the natural numbers will be called a *class-expression*, "[these are] ordered in a sequence… R(n)…of *class-expressions* and the ordering R can be defined in the formal system." "let A be a class expression… let {A:n} be the formula which arises by substitution of [the symbol for] the number n for the free variable. The ternary relation x = {y; z} is definable in the formal system … [z being a variable of type integer, y a variable of type class expression and x a variable of type wff] … Define a set K of natural numbers by

$$(\dagger) \qquad \mathrm{n} \in \mathrm{K} \ \Xi\ \neg\mathrm{Bew}\{\mathrm{R(n)} : \mathrm{n}\}$$

where Bew f means f is a provable [Beweissbar in German] formula. Since the concepts occurring in the definiens [right side of (†)] are all definable in the system, so also is K …i.e. there is a class expression S such that **{S:n} intuitively interpreted says that n belongs to K**. …S is identical with some R(q) … there is not the slightest difficulty in writing down the formula S."

"We now show that the formula {R(q): q} is undecidable.... for if it is assumed to be provable then it would be true... [i.e. {S: q} would be true] so that q belongs to K ... by (†) ¬ Bew {R(q): q} would hold, contradicting our assumption [that {R(q): q} is provable]. ...If the negation ¬ {R(q): q} is provable [second assumption], i.e. [¬ {S: q} is provable, hence true which means that q does not belong to K], that is, Bew {R(q): q} would be provable [hence true].which is again impossible [contradicting our second assumption]." "...there is a close relationship with the Liar paradox." "... {R(q):q} says that q belongs to K; i.e. by (†) {R(q):q} is not provable. Thus, we have a proposition before us which asserts [by its interpretation] its own unprovability...This method of proof can obviously be applied to every formal system which first possesses sufficient means of expression...to define the [system] concepts (especially the concept *provable formula*) and secondly in which every provable formula is true... we shall replace the second assumption by a purely formal and much weaker assumption."

Appendix 2

The Lambda Calculus

In Chap. 3, we presented Turing's theory of computation based on his machines as first introduced in his 1936 paper on the Entscheidungs (Decision) problem. As noted in several chapters, the Turing machine pioneered many ideas adopted in modern digital computers and is probably more important for that achievement rather than its original purpose of proving the unsolvability of the Decision problem. Recall that Turing's paper was preceded by a few weeks by a paper by the American logician Alonzo Church proving the same unsolvability result by a quite different method called the *lambda calculus*. In this appendix, we give a short summary of the lambda calculus. Besides its original purpose of defining an *effective procedure* to be used in proving the unsolvability of the Decision problem, it turns out that many ideas in the lambda calculus correspond to techniques employed in modern programming languages, for example techniques for substituting values for variables in procedure calls (See Chap. 4).

As described in Appendix 1, in the early years of the twentieth century there was a very active development of formal logic. The propositional calculus was a formal system developed to formalize informal mathematical proofs involving declarative statements with no variables. The predicate calculus was a formal system developed to formalize mathematical proofs involving statements containing predicate expressions, that is, expressions formed with individual variables. These logical calculi prescribed precisely formulas, called *well-formed-formulas* (*wff's*) to represent informal mathematical statements used in proofs of theorems and furthermore, specified precise rules for manipulating wff's in formal proofs. The *lambda calculus* is a formal system devised by Alonzo Church and his Princeton graduate students S.C. Kleene and J.B. Rosser in the 1930s and 1940s to represent mathematical functions by wff's called *lambda expressions* and to prescribe rules for manipulating such wff's in a manner that corresponds to the various ways functions are manipulated in calculations found in informal mathematical treatises. The lambda calculus was supposed to include all *effectively calculable* functions, that is, all mathematical functions that could actually be calculated by obviously executable means.

The concept of a mathematical function had been in general use by mathematicians for many years before the 1930s but, surprisingly, without a commonly accepted notation for functions and without precise rules for manipulating functions in calculations. Functions were often defined by algebraic formulas, for example by polynomials in one variable such as $x^2 + 2x + 1$. A typical phrasing could be as follows: "let f be a function of the variable x defined by $f(x) = x^2 + 2x + 1$". The notation "f(x)" was of fairly recent origin. An alternate common phrasing, still in use, could be: "let f be a function such that $f: x \to x^2 + 2x + 1$". This suggests that f is conceived of as a *mapping* from a *domain* set D into a *range* set R, stated as $f: D \to R$, where x is in D and f(x) is in R. Some texts included heuristic two-dimensional diagrams depicting areas D and R with arrows on directed paths drawn from D to R to indicate the direction of the mapping. It was understood that there could only be at most one directed path emanating from any point x in D and ending at a point y in R. These informal notations are adequate for functions f of a single variable which has a well-defined domain D and range R, for example subsets of the real numbers. We can then think of "applying" f to a value d in D to obtain a value f(d) in R. The procedure of function *application* was the main abstract procedure performed with functions. If f(x) is defined by an algebraic expression, E say, involving the variable x, then application is *effectively* executed by substituting d for x in the expression E and carrying out the indicated algebraic operations in E. We shall denote this substitution calculation by E $(x \to d)$ and define the application by

$$f(d) = E(x \to d).$$

For functions f of two or more variables, the defining expressions are more complicated and the process of applying f is more complicated. For example, let $f(x, y) = x + y$. We can apply f to an ordered pair (2, 3) to get the value f $(2,3) = 2 + 3 = 5$, if it is agreed implicitly that the ordering (2, 3) corresponds to (x, y). However, we can also apply f to (x, 3) to get $f(x, 3) = x + 3$, which is another function. So application of a function to *arguments* can yield other functions as results. Furthermore, mathematicians of that era were already considering higher-type functions. For example, for any integrable function f(x) the integral $I(f) = \int_a^b f(x)dx$ defines a function I of f which has a real numerical value I(f). Even more generally, in some mathematical contexts there arise functions F, called transformations, which map a function f onto another function g, so that $F(f) = g$. A formal system which aims to provide a calculus for specifying and manipulating arbitrary mathematical functions must take the higher-order functions like F into account. This is the ultimate goal of the lambda calculus. A concomitant goal of Church was to restrict the manipulations on function wff's to be "effective calculations" on symbol strings representing functions, where by "effective calculations" he meant such as would be acceptable to the Hilbert formalist school as parts of a decision procedure. However, he had to be careful that the restrictions would not limit the class of decision procedures by excluding some that, although complicated, should be clearly admissible. With the examples of the propositional and predicate calculus as a guide, Church devised the formal system called the *lambda calculus* as a system to provide wff's to represent arbitrary mathematical functions and to provide rules for formally manipulating these wff's in a manner which corresponded to the way functions are manipulated in informal procedural mathematics. If the wff's and rules of the lambda calculus were properly formulated, then Church hoped and hypothesized that an *effectively calculable* function would be one expressible as a wff in the lambda calculus and an *effective procedure* acceptable to the Hilbert formalists would be available in the manipulations of such wff's. When Turing proved in an appendix to his original 1936 paper that Turing machines and lambda calculus effective procedures are equivalent in that they yield the same class of procedures this supported Church's hypothesis and it became the *Church-Turing thesis* that all computable functions are given either by Turing machines or the lambda calculus.

Syntax and Rules of Derivation of the Lambda Calculus

The main ambiguities in traditional function notation were the designation of how variables are used as symbolic arguments in function expressions E and the rules for applying a function expression E to actual arguments so as to obtain a function value. The preceding examples illustrate this ambiguity problem in the simple case where E is an algebraic formula. It was the main problem addressed by the lambda calculus. To explain Church's solution, let us consider the syntax of lambda calculus wff's expressing functions. We denote the set of such function wff expressions by . We define it recursively as follows:

We assume a starting set of symbols called *variables,* say x, y, z etc..

1. Certainly a variable x is in ^ (These are the simplest function wff's).
2. To obtain more complex function expressions, say using more variables, we suppose that E and F are in ^ . Then the *application* of E to F, denoted by the concatenation (EF) is also in ^ (Parentheses can be omitted since application is assumed to be left-associative, so that EFG means (EF)G. Thus, xyz is a wff involving three variables).
3. Let E be in ^. Let x be a variable. Then the expression (λx.E) is in ^. The operator λ is called the *abstraction* operator (It serves to designate possible variables x in E for which substitutions of "values" are to be made in evaluating E. It makes explicit which variable enters into an *application* procedure as stated in the beta-conversion procedure below. The *scope* of the abstraction operator is defined to be maximal so that λx.EF means λx. (EF) and not (λx. E)F. For multivariate functions we use λxλyλz E abbreviated as λxyzE. λx.E *binds*

the variable x in E. All other variables in E that are not bound by an abstraction are *free* with regard to substitution. A variable is bound by its nearest abstraction operator).

There are two symbol-manipulation rules of the lambda calculus.

Alpha-conversion: change a bound variable x in λx.E to any other variable y to obtain λy.E as long as y is free, that is, not already bound, in E (This makes clear the role of bound function variables as place holders).

Beta-conversion (*or reduction*): Let E and F be lambda expressions in ^. Suppose E contains the variable x free and F does not. Then λxEF can be *converted* (*reduced*) by substituting F for x in E to get E(x → F) (This rule formalizes the application of a function to a specified argument and the evaluation of the result. Its inverse applied to E(x → F) yields the abstraction).

As a simple but interesting example, consider λx. x. Which function does it represent? Just apply it to any expression F and use beta-conversion to get (λx.x) F = (x → F) = F. So this represents the identity function.

Example: Suppose that the integers 2 and 7 have been encoded as lambda expressions (See below). Let the product 2*n also be encoded as a lambda expression. Then

$$\left(\lambda \mathrm{n}.2^{*}\mathrm{n}\right)7 = 2^{*}7 = 14.$$

This illustrates how symbol manipulation occurs in the lambda calculus formal system. Since every lambda expression denotes a function, in order to deal with domains like the integers some lambda expressions must be used to represent the integers n = 0, 1, 2, ... Church suggested that an integer n be represented by the n-fold composition f^n of any function f with itself. Here $f^2(x) = f(f(x))$ and f^n $(x) = f(f(\ldots f(x))\ldots)$ with n factors f. Also f^0 is defined to be the identity function. Specifically he defined

$$0 ::= \lambda \mathrm{fx.x},\ 1 ::= \lambda \mathrm{fx.fx},\ 2 ::= \lambda \mathrm{fx.f(fx)}),\ 3 ::= \lambda \mathrm{fx.f(f(fx))},\ \text{etc.}$$

Applying these lambda expressions to an expression E which does not contain f and using beta-conversion, we get 0E = λf.λx.xE = λf.(x → **E**) = **λf.E** = E; 1E = λf. fE = fE = f(E); 2E = λf.f(fE)) = f(fE) etc. as in ordinary composite function notation. The exponential behavior of n in f^n yields the definition of addition, mPLUSn, namely, $f^{m+n} = f^m\ f^n$. Then multiplication MULT is defined as repeated addition. Thus, mMULT n means add up n m times. A lambda expression which abbreviates multiplication is

```
MULT ::= λmn.m (PLUS n) .
The power or exponent function POW for integers is
abbreviated by
     POW ::= λbe.e b .
```

To facilitate the use of the rather abstract symbolism in the lambda calculus it is convenient to introduce other abbreviations. By convention, the following two abbreviations (known as *Church booleans*) are used for the boolean values TRUE and FALSE:

```
TRUE ::= λxy.x
FALSE ::= λxy.y
```

(FALSE is equivalent to the Church numeral zero, the identity function, defined above)

Then, with these two abbreviated λ-expressions, we can define some basic logic operators as follows:

```
AND ::= λpq.p q p
OR ::= λpq.p p q
NOT ::= λpab.p b a
IFTHENELSE ::= λpab.p a b .
```

To verify that these expressions yield the usual Boolean values, consider the Church encoding for pairs, PAIR, which encapsulates the ordered pair (x,y) and the abbreviations FIRST for the first member x and SECOND for the second member y of a pair. By beta conversions it can be verified that FIRST returns the first element of the pair, and SECOND returns the second.

```
PAIR::= λxyf.f x y
FIRST ::= λp.p TRUE
SECOND ::= λp.p FALSE
NIL ::= λx.TRUE
  NULL : := λp.p (λxy.FALSE)
```

By applying the abbreviated logical operator expressions for AND etc. to the abbreviation for a (TRUE FALSE) pair the reader can verify that the logical operators satisfy the usual truth tables. Thus, for example, by beta conversion,

AND (TRUE FALSE) → FALSE .

For numerical functions, the following expression defines the predicate ISZERO. Just apply it to any integer n as defined above to verify that ISZERO n is TRUE for n = 0 and FALSE for nonzero n.

```
ISZERO ::= λn.n (λx.FALSE) TRUE .
```

By setting up such abbreviations for familiar mathematical and logical constructs, one can become familiar with the use of the lambda calculus as a formal system for manipulating the usual mathematical and logical functions and apply it to the decision problem which deals with proofs in predicate calculus. Rather than pursue this strategy further to obtain Church's unsolvability result, we shall now consider other concepts of the lambda calculus which have been important for programming languages.

A structure that is useful in computer programming is the *linked list* or simply *list*. A list of elements (such as integers) can be defined as either NIL for the empty list, or the `PAIR` of an element and a smaller list. The predicate `NULL` tests for the value `NIL`.

Another computer programming structure is the *procedure abstraction* which serves as a subroutine in procedural languages like Fortran, C and C++. As noted in Chap. 4, Turing's 1936 paper introduced subroutines as separate program sections P outside the main program having identifying names, such as P(x, y), where x and y are variables in statements in the defining *body* of P(x,y). The variables serve as parameters to be replaced by expressions u and v designated in a *call* statement, having the format P(u, v), at some point in the main program. The semantics of a *call* can be defined in the lambda calculus in terms of abstract function application, that is, as an application of P(x,y) to (u,v). We mentioned two possible useful cases of u and v, namely, as data values in a *call-by-value* and as addresses in a *call-by-name*. Other cases are possible. To define them formally, Church defines a predicate which determines whether a given lambda expression has a *normal form.* A normal form is an equivalent expression which cannot be reduced any further under the rules imposed by the form. Then he assumes that this predicate is computable, and can hence be expressed in lambda calculus. The term "redex" stands for an expression reducible by alpha or beta conversion. The latter may involve different orders of application. The lambda calculus considers various orders which have analogs in programming language when considering procedure bodies and procedure calls. For example, a procedure body P(x,y) can contain calls to other procedures Q(x, y). What substitutions should be done in such a call to Q(x, y) to execute a call P(u, v)? The lambda calculus offers several possible orders.

Applicative order
The leftmost, innermost redex is always reduced first. Intuitively this means a function's arguments are always reduced before the function itself. *Applicative order* always attempts to apply functions to normal forms, even when this is not possible.

Most programming languages (including Lisp and C and Java) are described as "strict", meaning that functions applied to non-normalising arguments are non-normalising. This is done essentially using applicative order in a call-by-value reduction (see below), usually called "eager evaluation".

Normal order
The leftmost, outermost redex is always reduced first. That is, whenever possible the arguments are substituted into the body of an abstraction before the arguments are reduced.

Call by name
As in normal order, but no reductions are performed inside abstractions. For example `λx.(λx.x)x` is in normal form according to this strategy, although it contains the redex `(λx.x)x`.

Call by value
Only the outermost redexes are reduced: a redex is reduced only when its right hand side has reduced to a value (variable or lambda abstraction).

As pointed out by Peter Landin's 1965 paper “A Correspondence between ALGOL 60 and Church's Lambda-notation”, *Commun. ACM 8,* 89–101, 158–165*,* sequential procedural programming languages can be defined in terms of the lambda calculus, using the basic mechanisms for procedural abstraction and procedure (subprogram) application as enumerated above.

Turing's Proof of the Unsolvability of the Decision Problem

Since we have not presented Church's lambda calculus proof of the unsolvability of the decision problem for predicate calculus, we shall give a very brief outline of Turing's machine-oriented proof. Turing follows Godel in setting up numerical codes for arbitrary Turing machine programs and their states and numerical functions for state transitions. The code or *description number* for M is computed from its program and is denoted by SD(M) (See Chap. 3). Conversely, given SD(M) one can reconstruct the program for M. He then constructs a predicate calculus wff U(SD(M)) for an arbitrary machine M that contains SD(M) as a subterm and which can be interpreted as asserting informally that there exists a configuration state q of M and a tape square n which contains the integer 0 when M is in state q. Formula U is built from several predicate wff's which formalize how the program for machine M causes M to execute its computation steps. Thus, Turing arithmetizes his machines M and their computations.

He uses a formula for the integer successor function, SUCC(n,m), (interpreted as asserting that $m = n + 1$) and the predicate formula INT(n) which is interpreted as true whenever n is an integer. Then he defines SD(M) by Godel's technique of assigning integer values to the symbols in the program of instructions for machine M. He constructs a predicate K(q, s) which asserts that q is the internal configuration of M when the total state of M is s. This is a straightforward coding calculation from s. The predicate I(s, k) is true if in state s the tape square k is scanned. This is again obtained from the definition of state s. The formula F(s, t) is interpreted as true if state t is a possible successor of state s according to the program for M. Finally, the predicate R(s, k, x) is true if for state s the scanned square is number k and contains the character x. Again this predicate can be constructed from the definition of the state of M as explained in Chap. 2. The wff U is a rather long wff which uses the above predicates and functions. We break it up into four lines as follows:

$(\exists n)[\mathrm{INT}(n)\ \&\ \Delta(m)\ (\mathrm{INT}(m) \rightarrow \exists y F(m,y))$
$\&\Delta v \Delta z(F(v,z) \rightarrow \mathrm{INT}(v)\&\mathrm{INT}(z)\&\Delta m R(n, m, w)$
$\&I(n,n)\&K(q,n)\&SD(M)]$
$\rightarrow \exists s \exists m(\mathrm{INT}(s)\&\mathrm{INT}(m)\&R(s, m, 0).$

The fourth line carries the main interpretation of U. It states that there exists a state numbered s of M in which the scanned square is numbered m and contains the character 0. The first three lines assert that there is a number n (zero) which is the number of the first state and the first scanned square and further for each state m there is a successor state y.

Turing first proves (see Chap. 3) that there is no machine which can determine whether any machine M is circle-free and consequently whether M ever prints 0. He then establishes two Lemmas that U(SD(M)) is provable if and only if the character 0 appears on the tape of M in some state s. Next he assumes the Decision problem is solvable by some Decision machine TD. This means that TD can decide whether any wff, in particular any wff like U(SD(M)) for any M, is provable. But by the Lemmas, TD can then decide whether 0 appears on the tape of M. So TD can decide whether any M ever prints 0. But he has already shown that there is no such machine procedure. It follows that that there is no such machine TD. This contradicts the assumed existence of TD. So the Decision problem is unsolvable.

Chapter 4
The Software Side of Computer Science – Computer Programming

Edward K. Blum and Walter Savitch

Still following Turing's fundamental conception of Computation as having two indissoluble complementary sides, namely, *hardware* and what we now call *software*, as implied by the ideas in his groundbreaking 1936 paper and other papers, in this Chap. 4 we give a description of the *Software side* of Computer Science. We present the hardware side in the next chapter. We choose this ordering of presenting the two sides because, as we shall explain in the next chapter, the conception and design of a new computer begins mainly with considerations of its software as defining its major functions and then continues with the hardware elements providing the physical devices to implement these functions.

For the software side, we focus on *computer programming*, the subject that deals with *programming languages*, the languages used to specify computations. A comprehensive exposition of this subject could easily fill a lengthy book. Indeed there are several books which cover the details of specific computer programming languages. We shall discuss the example of the Fortran language below so that the reader will have some familiarity with *high-level* programming. In this Chapter, we do not attempt a complete coverage of Fortran or any programming language and intentionally omit most of the syntactic aspects of such languages. Here, we shall mainly discuss the *semantics* of programming languages, that is, the meanings of the main language constructs that have been used traditionally in most programming languages and in fact are in current use. However, for completeness, key aspects of the syntax of programming languages are discussed briefly in the Appendix to this chapter in connection with a syntax specification technique called BNF notation.

E.K. Blum (✉)
Department of Mathematics, University of Southern California, Los Angeles, CA, USA
e-mail: blum@usc.edu

W. Savitch
University of California, San Diego, La Jolla, CA, USA

E.K. Blum and A.V. Aho (eds.), *Computer Science: The Hardware, Software and Heart of It*, DOI 10.1007/978-1-4614-1168-0_4,

Strictly speaking, the *Software side* overlaps what is now called *Theoretical Computer Science*, which we regard as the *Heart* of Computer Science, and which we have already presented in Chap. 3 and Appendix G of Chap. 3 (e.g. Computability, Undecidability and Unsolvability). These theory topics are fundamental to both the Hardware and Software sides of Computer Science, and were shown to determine the boundaries of what is possible on both sides. Logically, it was reasonable to treat these computer science theory topics first so as to understand the limits of the two main sides. Other parts of Theoretical Computer Science (e.g. complexity theory, data base theory, parallel-distributed computing, etc.), will be presented in later Chapters after we have laid out the constituents of the Hardware and Software sides and thereby obtained a perspective of these basic interrelated constituents which form the edifice of the whole subject. On the Software side, clearly programming languages are a basic constituent. Therefore, we now proceed to consider the computer programming elements of the Software side.

As we have shown in Chap. 3, in his 1936 paper Turing devotes considerable space to explaining the syntactic structure of his machines' instructions. These instructions are the basic statements in programs for Turing machines. Indeed, they formally define the machines and specify how they compute, along with some informal natural language specifications. However, many variants of the syntactic details are clearly possible. What really matters is that the instructions are available to a computer user to specify the *semantics* of Turing machines; i.e. how a machine executes simple basic operations on the machine's tape. (In a later paper, E. L. Post simplified the basic operations further.) Turing also discusses the effect of executing in sequence a finite list of instructions which are to implement a particular machine, for example an adding machine to add any two integers. Such a finite list of instructions is an early example of what we now call a *computer program*. Furthermore, to prove his paper's main result (that Hilbert's decision problem is unsolvable) he invents a universal machine. A *universal* Turing machine is a major original idea (in those early days of 1936) of a machine that can have access on its tape to the programs which specify other machines, such as an adding machine. By "reading" a particular machine's program *stored* on its own tape, a universal machine can execute that particular machine's computation. Thus, a universal Turing machine is the first general example of a *stored-program* computer. Babbage's *analytic engine* design also suggested that programs be stored in memory along with data, but did not exploit the idea. Execution of stored programs is the most powerful capability of the real computers treated in Chap. 5 and of all modern *digital* computers. (The meaning of the term *digital* will be made clear in the hardware chapter. Essentially, it connotes that the data is in digital format represented by discrete characters such as the digits 0,1,..,9, and letters A. . .Z.) The stored-program concept is what makes Computer Science an inseparable union of its Hardware and Software Sides.

Furthermore, in his software development, Turing attempts to simplify the description of the subtle and complex program sections that make up his universal machine U. Turing invents schematic programs which employ parameters represented by variables. These program *schemes* perform certain useful functions on their variables and can be *called* on by statements in the *main program* for U, the syntax of which allows subroutine calls as well as the simple basic quadruple instruction statements. In effect, these program schemes were the first *subroutines* in the Software history of Computer Science. Much effort has been expended on the syntax and semantics of subroutines in later research on computer programming. To describe this research, it is useful to do a short review of some of the major steps in the evolution of programming as a computational activity. We shall skip the early ideas of Lady Ada in programming the Babbage analytic engine referred to in Chap. 2. We jump to the 1940s and thereafter and summarize some well-known steps in the later evolution of programming concepts and languages.

Relay calculators such as the CPC (Card Programmed Calculator) were programmed in algebraic notation, allowing formulas such as A + B to be punched on cards, which cards were read by a card reader machine so that the punched out holes allowed electrical actuation of relays to cause execution of the + operation. Similarly, formulas were used to program the earlier electromechanical desk calculators (e.g. the Marchant calculators) which allowed (required) human users to punch a key for +. Of course, the numerical data values of A and B had to be input directly as calculator punched code or *assigned* previously on other cards in the CPC. These data values were stored in registers suitably addressed for identification of variables like A and B.

With the advent of electronic computers such as those to be described in Chap. 5, employing electronic memories with many addressed cells to store data values of variables like A and B, such variables came to be regarded as the *addresses* of the memory cell locations. Various experiments with computer hardware instruction formats investigated single-address, two-address and even three-address instructions. These formats were reflected in a low-level programming *assembly* language of *statements* (such as C = A + B for a three-address instruction) which could be written by a programmer and input to and read directly by the hardware and executed by the machine as explained in Chap. 5. However, it soon became evident that there were many problems with such a *low-level assembly programming language* for even simple computations. By forcing the human programmer to stay close to the low machine-level operations on addresses, assembly language programming was much too involved in machine-oriented details having to do with memory organization that were far removed from the mathematical formulas which originally defined the computation. Writing assembly language programs was a painfully slow process and subject to many errors. What was needed by human programmers was a *higher–level* language, that is, one closer to mathematical language, on a level above the machine level, which could be written more easily by the programmer familiar with the mathematical formulation of the computation. These higher-level language statements could then be translated (or as we now say *compiled*) by a computer into an assembly-like language for communication with the hardware. This led to much research and development of the structure of programming languages and such tools as *compilers,* the programs which *compile* mathematical language programs into machine assembly language.

For scientific computation problems, the Fortran language promoted by IBM (with project leader John Backus) became the higher-level language of choice in the 1950–1960s. For business applications, the COBOL language became the language of choice. Fortran went through several improved versions over the years. At about the same time, the high-level Algol language was promoted by the international programming community.

Just as Alan Turing can be regarded as the father of much of today's Computer Science, so can John Backus be regarded not just as the father of Fortran, which indeed he was, but also as the father of many of today's computer programming features. Since the state of the Computer Science "Union" is as much about the scientists who developed it as about the subject itself, it is appropriate and informative to give at this point a short biography of Backus. Much of the information recorded here is drawn from the Backus obituaries which appeared in the IEEE Spectrum issue, March 20, 2007 and in the N.Y. Times, March 19, 2007. However, the author of this Chapter had the privilege of knowing Backus personally and could not refrain from injecting some personal observations. The author met Backus when both were students studying mathematics at Columbia University in the early 1950s. As far as is known, neither had any overt interest in computers or computation. Computers were then a new phenomenon, but IBM maintained a computer lab not far off campus and legend has it that Backus, out of curiosity, found his way to that lab and eventually to a position at IBM, where he remained all his working life. Their paths crossed in later years on many happy occasions at conferences while the author was Secretary of IFIP Working Group 2.2 (Formal Description of Programming Concepts) and Backus was an active member.

John W. Backus was born in 1924 and died in 2007 at his home in Ashland, Ore. He was 82. His daughter Karen Backus announced the death, saying the family did not know the cause. Backus assembled and led the I.B.M. team that created Fortran, the first widely used high-level programming language. It was released to the public in 1957, which event many consider to be a giant step forward in the history of computer software comparable to the giant step forward in hardware development that occurred when the micro- processor chip was introduced. Fortran changed the mode of communication between humans and computers, moving it up several levels above machine level by providing a language that is more comprehensible by humans and fairly free of machine-oriented language constructs. Fortran is considered the first successful high-level language. Backus and his team, then all in their 1920s and 1930s, devised the Fortran programming language as a combination of English shorthand and algebraic expressions. Ken Thompson, who developed the Unix operating system at Bell Labs in 1969, observed that "95% of the people who programmed in the early years would never have done it without Fortran." He added: "It was a massive step forward."

Backus realized that to be successful in practice Fortran programs had to be efficient in execution, running as fast as assembly language programs hand-coded by expert programmers who worked with machine-oriented assembly languages. Adequate efficiency was achieved by the excellent design of Fortran *compilers*, programs which translated Fortran programs into executable machine-compatible assembly language programs.

Backus grew up in an affluent family in Wilmington, Del., the son of a stockbroker. In a series of interviews in 2000 and 2001 in San Francisco, where he lived at the time, Backus recalled that his family had sent him to an exclusive private high school, the Hill School in Pennsylvania. "The delight of that place was all the rules you could break," he recalled. After flunking out of the University of Virginia, Backus was drafted in 1943. His scores on Army aptitude tests were so high that he was assigned to training programs at three universities, for studies ranging from engineering to medicine. After the war, Backus became a student at Columbia University in mathematics, receiving his master's degree in 1950. Shortly before he graduated, Backus wandered by the I.B.M. office in New York, where one of its electronic calculators was on display. When a tour guide inquired, Backus mentioned that he was a graduate student in math; he was taken upstairs and asked a series of questions which he described as math "brain teasers." He was hired on the spot. " "As what?" someone asked. "As a programmer," Backus replied. "That was the way it was done in those days." The first written reference to "software" as something distinct from hardware probably did not come until 1958.

In 1953, frustrated by his experience with low-level programming, described as "hand-to-hand combat with the machine," Backus decided that he would like to do research to simplify programming. He wrote a brief note to his IBM superior, asking to be allowed to head a research team with that goal. "I figured there had to be a better way," he said. Backus got approval and began hiring until the team reached 10 individuals. It included a crystallographer, a cryptographer, a chess wizard, an employee on loan from United Aircraft, a researcher from the Massachusetts Institute of Technology and a young woman who joined the project straight out of Vassar College. "They took anyone who seemed to have an aptitude for problem-solving skills – bridge players, chess players, even women," said Lois Haibt, the Vassar graduate. Backus, colleagues said, managed the research team with a light hand. The hours were long but informal. Snowball fights relieved lengthy days of work in winter. I.B.M. had a system of rigid yearly performance reviews, which Backus deemed ill-suited for his programmers, and so he ignored it. "We were the hackers of those days," Richard Goldberg, a member of the Fortran team, recalled in an interview in 2000.

As part of his broader interest in programming languages, Backus developed, with Peter Naur, a Danish computer scientist, a notation for describing the syntactic structure of programming languages. It became known as Backus-Naur form or BNF and was a standard tool for specifying the syntax of programming languages. (See the Appendix on BNF below.)

Later, Backus worked in an area called *functional programming*. That effort, Backus said, was to develop a system of programming that would focus even more on describing the problem a person wanted the computer to solve and still less on giving the computer step-by-step instructions. However, it would seem that some elements of machine-like instructions are necessary for a practical programming language. How did Backus's team develop and improve Fortran? Innovation, Backus said, was a constant process of trial and error. "You need the willingness to fail all the time," he said. "You have to generate many ideas and then you have to work very hard only to discover that they don't work. And you keep doing that over and over until you find one that does work." Backus once said: "Much of my work has come from being lazy. I didn't like writing programs, and so, when I was working on the IBM 701 writing programs for computing missile trajectories, I started work on a programming system to make it easier to write programs."

Backus and his colleagues, over several decades, updated Fortran on numerous occasions. Improvements included the addition of support for processing of character-based data, array programming, module-based and object-based programming, and object-oriented and generic programming. A recent edition of the language, Fortran 2003, is a major revision that introduces many new features. The legacy of Fortran is far reaching. Even today, half a century later, floating-point benchmark programs to gauge the performance of new computer hardware are still written in Fortran. This implies that the new computers had Fortran compilers.

Backus spent his entire career at IBM. In 1987, the company named him an IBM Fellow. In addition to the prestigious McDowell Award, he was recognized by: the National Science Foundation (on behalf of the U.S. Congress) with the Presidential Medal of Science in 1975; the Association for Computing Machinery with the A.M. Turing Award in 1977; and the National Academy of Engineering with the Charles Stark Draper Prize in 1993.

Appendix: The BNF Notation, Syntax of Programming Languages

As stated earlier in this Chapter, we wish to de-emphasize the discussion of the syntax of programming languages. However, we cannot ignore it entirely. We now discuss syntax by explaining the BNF notation. BNF is so widely used in specifying the syntax of programming languages that we must at least devote this short Appendix to it. Furthermore, it is close to the *grammars* used in formal language theory to specify context-free languages. Formal language theory is a traditional important part of Theoretical Computer Science and is not covered in this book, but from the following brief account of BNF the reader can gain some insight into the ideas and methods treated in that theory.

BNF is an acronym for "Backus Naur Form". Backus and Peter Naur introduced for the first time a formal notation to describe the syntax of a given language. This was for the description of the ALGOL60 programming language. It seems that most of BNF was introduced by Backus in a report presented at an earlier UNESCO conference on ALGOL 58. Few read the report, but when Peter Naur read it he was surprised at some of the differences he found between his and Backus's interpretation of ALGOL 58. He decided that for the successor ALGOL 60, the syntax should be given by a precise formal method so that all participants in future meetings on ALGOL 60 would be aware of what changes they were agreeing to. He made a few modifications and drew up on his own the BNF grammar for ALGOL 60. So BNF was introduced by Backus in 1959 and by Naur in 1960. (For more details, see the introduction to Backus' Turing award article in Communications of the ACM, Vol. 21, No. 8, August 1977.)

Since 1960, almost all books on particular programming languages use BNF to specify the syntax of the languages. We give an outline of **BNF** notation and some examples of its use.

There are three main *metasymbols* used in statements in BNF as follows: **::=** | < >

::= means "is defined as" in the statement of a grammatical rule, called a *production*.
| means "or"
< > angle brackets are used to surround syntactic category names.

The angle brackets distinguish syntax class names (also called *non-terminal* symbols) from *terminal* symbols which are written exactly as they are to be represented in the language.

A *grammar* for a language consists of *productions,* also called *rules,* which display the linear format of a syntactic class. For example, a BNF rule defining a nonterminal has the following format:

<nonterminal> ::= sequence_of_alternatives, each alternative consisting of strings of terminals or nonterminals, alternatives are separated by the meta-symbol |

As another example, the BNF production for a *program* in some example mini-language is

```
<program> ::= program <declaration_sequence>
        begin
        <statement_sequence>
        end ;
```

This displays that in the mini-language, a program consists of the terminal keyword "program" followed by a declaration sequence, then the keyword "begin" and a statement sequence, finally the keyword "end" and a semicolon.

This contrived example language program is not too far from real languages.

Optional items in BNF definitions are enclosed in square brackets meta symbols [and], for example:

```
<if_ statement> ::= if <Boolean _expression> then
 <statement _sequence>
 [ else
  <statement _sequence> ]
  end if ;
```

Thus the else clause in an if-then-else _statement is optional. The semantics of this standard statement found in most programming languages is clear. It specifies a *branch* point in the list of program statements. The Boolean _expression is evaluated and if the value is True, then the first statement _sequence is executed. If the value is False, then the first statement _sequence is skipped and the *branch* given by the statement _sequence following the else is executed. The program then continues with statements after the if-then-else statement. Repetitive items (zero or more times) are enclosed in metasymbols braces { and },for example:

```
<identifier> ::= <letter> { <letter> | <digit> }
```

This rule specifies that an identifier (usually used as a program variable) is a letter followed by zero or more letters or digits; e.g. x, x1, y, xy, z21. This rule is equivalent to the *recursive* rule

```
<identifier> ::= <letter> | <identifier> [ <letter> | <digit> ]
```

Terminals of only one character are surrounded by quotes (") to distinguish them from meta-symbols, for example,

```
<statement_sequence>::= <statement>{";"<statement > }
```

defining a sequence of statements as being separated by semi-colons.

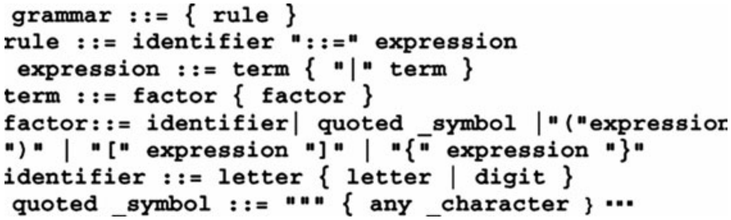
In recent text books, terminal and non-terminal symbols are distinguished by using bold faces for terminals and suppressing < and > around non-terminals. This improves the readability. The preceding example then becomes:

```
if_statement::=if  Boolean _expression
 then
  statement _sequence
 [ else
  statement _sequence  ]
  end if ";"
```

One of the most important statements besides the *conditional* if-then-else statement in Fortran and many other programming languages is the *assignment* statement which assigns a new value to a variable. The variable is on the left side of an = sign and can be a simple identifier like X (see below) or have a more complex structure, say X(i, j), denoting an array entry. The new value, written on the right side of the = sign, is defined by an *expression* (see below) specifying a numerical evaluation based on previously computed values such as (A + B) or a Boolean evaluation like A OR B, where A and B have been previously assigned values. Thus, in BNF,

```
assignment_statement::=<variable>"="<expression>";"
```

Now as a last example here is the definition of a BNF grammar expressed in BNF:



```
 grammar ::= { rule }
rule ::= identifier "::=" expression
 expression ::= term { "|" term }
term ::= factor { factor }
factor::= identifier| quoted _symbol |"("expressior
")" | "[" expression "]" | "{" expression "}"
identifier ::= letter { letter | digit }
 quoted _symbol ::= """ { any _character } """
```

BNF is not only used to state syntax rules but it is commonly used (with variants) by syntactic tools. See for example LEX and YACC, the standard UNIX *parser generators*. If you have access to any Unix machine, you will find there a description of these tools. (Also see the Johnson reference below.)

Other Programming Languages

Many other programming languages were developed after the pioneering Fortran. The International Federation of Information Processing (IFIP) organized various working groups to investigate many software topics. (See the Appendix below for a list of some well-known languages developed after Fortran.) For example, IFIP Working Group 2.1 fostered research on the Algol language as a sequel to Fortran, but IBM managed to keep Fortran at the forefront of popular usage until the C and C++ languages were developed at Bell Labs. (See Appendix below.) With its well-conceived combination of machine-like and mathematical statements, C and then C++ became the language of choice. C introduced programming constructs like *pointers*, which are variables which have memory location values that can be manipulated to construct memory structures like the linked lists made popular in the LISP language (project leader John McCarthy) which was employed by researchers in artificial intelligence, which incidentally was a research topic of interest to Turing, who raised the question "Can machines think?". He wrote a paper on it and formulated the Turing Test which a computer must pass to be judged as a successful *thinking mechanism.* Othe**r** related research, on brain modeling, was done at M.I. T. by the neurophysiologist Warren McCullough and his mathematician co-worker Walter Pitts. McCullough and Pitts used Boolean algebra in their conceptual models of neurons and neural networks. This idea was actually implemented in a machine called a Perceptron by Frank Rosenblatt, but with limited success. The human brain still defies any computer-based modeling to explain its complex behavior. Recall Chap. 3 and our remarks on the Church-Turing thesis. Also see chap. 14 on quantum computing.

C++. This is a language which employs many of the basic Fortran statements such as assignments, if-then-else statements etc. Like C, it also employs machine-oriented constructs like pointers. However, C++ and the related JAVA language take a radical departure in viewpoint called *object-oriented* programming (OOP). OOP introduces the concept of a *class.* A *class* can be viewed, by using earlier research concepts, as an *abstract data type*, that is, as an algebraic structure consisting of abstract elements called *objects* and *operations* on objects. In JAVA, the operations are called *methods.* (We shall not take the space to elaborate on the classes on which OOP is founded. As a reference we cite the book "JAVA An Introduction to Computer Science & Programming- second edition" by Walter Savitch, Prentice Hall 2001.)

Parallel Computing Languages

As parallel computers came into use, especially by the military, a new set of programming constructs was needed to specify parallel computations. In the 1970s the U.S. military sponsored the design and development of a new language called *Ada*, which incorporated new constructs for specifying *parallel computation*. However, Ada was never widely adopted outside the military. Instead, *cluster computing* (on networked clusters of PC's) became the paradigm for parallel distributed computing and a new standard of programming constructs (called MPI for *Message Passing Interface*) was specified by committee. See Chap. 7 on High Performance Computing and Communications (HPCC).

Object oriented languages **(e.g. C++** and Simula) and network languages (JAVA) are more recent developments**.** There are several textbooks on C++, for example the books "Problem Solving with C++ The Object of Programming" by Walter Savitch, Addison-Wesley Pub. Co., Reading, Mass., 1996 and "C++ in Plain English" by Brian Overland, Henry Holt Co., N.Y. 1996. As noted above, like Fortran and other high-level languages, C and its successor C++ still provide the basic assignment statement to assign values to variables. Thus, in BNF,

Assignment _ statement ::= variable = expression ;

For example, X = a + b; assigns the value of the sum a + b to X. The equal sign does not mean "equal" as in mathematical equations. Rather X = should be read as "assign value to X", (or "store value in X") where X denotes a storage location in computer memory *declared* by a previous *declaration* statement about the identifier X. So one can legally write X = X + 1 to cause the value in X to be incremented by 1 and stored back in X. Most major steps in a computation are specified by *assignment* statements. The steps are executed in a linear sequence path as given by the order of the list of statements in a program, as in a Turing machine program. A *two-way branch* in the sequential path is specified by the *conditional* **if-then -else** statement mentioned earlier. This is a key statement in all modern programming languages and presumes that the hardware *Control* unit can execute a two-way branch in the sequential execution path. (See next Chapter.) Further *control* of execution steps is afforded by iterative loop statements of the form

while Boolean _condition { statement _sequence}

where the statement _sequence is executed repeatedly as long as the Boolean condition is evaluated as True. This is a variation of Fortran's iterative control statement

For boolean **Do**_loop,

where loop is a statement_sequence to be executed repeatedly as long as the boolean formula is True. The daring Fortran pioneers also used **GoTo** statements that allowed arbitrary **jumps** to labeled statements almost anywhere in the program. This was a source of many errors in sequence control specification and was eventually abolished. A restricted but sufficient control method using if.-then-else for branching and while statements for loops led to a style called *structured* programming.

Subroutines

As in Fortran and in Turing machines, C++ uses subroutines to allow the programmer to define functions that will be *called* (i.e. evaluated) at several places in the program with different values of parameters as arguments simply by stating the name of the subroutine with appropriate values for the subroutine parameters. Subroutine parameters are of two kinds, *call-by-value* with the usual obvious meaning of substituting a data value given in a call statement for the subroutine parameter and *call-by-reference* which substitutes a variable given in the call statement for the parameter, thus allowing the value of that variable to be changed by the subroutine execution. Further, by defining a function using *procedural abstraction* a programmer need only write comments outside the body of the function definition (called a *procedure*) that specify the required types of the parameters prior to a call to the procedure and specify the type of the value computed. These specifications are placed in a procedure declaration called a *prototype* at the "front end" of a program. The procedure definitions are usually placed at the back end of the program. The philosophy of procedural abstraction is extended in C++ to allow the definition of the aforementioned *abstract data types* which can be thought of mathematically as an algebra of *objects* and *functions* (i. operations) on the objects. An *object* is (represented by) a variable used in the function definition. An abstract data type is called a *class*. As noted above, programming with objects and classes is called *object-oriented* programming and is a powerful technique.

A class is defined using a class declaration. As an example we use the DayofYear class given in the Savitch book (page 292). It has one function called *output* which has no arguments and simply outputs a month and day to the monitor screen. When accessed by a call to its function *output*, this class produces a specified dialog between the programmer and the monitor screen using the built-in C++ stream functions **cout** to send out messages to the screen and **cin** to input values from the screen as typed by the programmer. The C *data structures* (*structs*) declared as the identifiers today and birthday (these structs declarations not shown here) are used to store day and month values. A struct consists of one or more *fields* of data separated by dots. The class function *output*, defined below, is then applied to the structs *today* and *birthday* (using the dot notation) to print the month and day fields of *today* and *birthday* to the screen. Note that the integer variables *month* and *day* are the specified objects operated on by *output*.

```
    class DayofYear
     {void output();
      int month; int day;}
  int main()
{ DayofYear today, birthday;
 cout<< "Enter today's date:\n";
 cout<< "Enter month as number:";
 cin>> today.month; //date input by user to the field month of struct today
 cout << "Enter day of the month:";
 cin >> today.day;
 cout<< "Enter your birthday:\n";
 cout<< "Enter month as a number:";
 cin>> birthday.month;
 cout<< "Enter the day of the month:";
 cin>> birthday.day ;
 cout<< "Today's date is";
   today.output();  // see definition of output below
 cout<< "Your birthday is";
 birthday.output();
    if (today.month = birthday.month && today.day = birthday.day
 cout<< "Happy Birthday \n";
 else cout<< "Sorry no birthday\n";
 return 0; }

void DayofYear::output()
  { cout<< "month = "<< month <<", day = " <<day << end; }
```

A Footnote

On the software side, there have been many interesting and notable programming languages besides the main ones described above (Fortran, C, C++, and Java), but space limits do not permit us to discuss them. Below we give an incomplete list of notable languages and their main developers and approximate dates of usage, as presented in the reference cited.

A List of Notable Programming Languages

1. Fortran John Backus, 1954
2. Lisp, John McCarthy, 1958
3. Algol 60, 1960
4. Cobol, 1960
5. Pascal, Ncholas Wirth, 1968
6. C Dennis Ritchie, 1968
7. Simula-67 O-J Dahl and K. Nygaard, 1967
8. Smalltalk, Xerox PARC, 1970's

9. CLU, 1970's
10. C++, 1980's
11. JAVA, 1990's

Appendix: Java Applets, HTML and the Web

The Java programming language is similar to C++ but includes special features which allow it to be used to write programs that can be run by accessing a document that is a webpage on a website on the WorldWideWeb (the "web"). As explained earlier, webpages are documents, treated as information resources, situated at websites having addresses (URL's) which locate them on the Internet. Websites are accessible by using a program called a web browser, such as Microsoft's Internet Explorer or Apple's Safari for example. These browsers use the HTTP protocol.

A web page document is written in the HTML language. A Java program can be embedded in an HTML web document and is called an *applet*. In this brief chapter, we sketch the HTML language and give an example of an applet to illustrate how webpages can provide a vast library of resources, both data and application programs (applets), of different types that can be utilized by computers and cell phones worldwide. Readers who have already used smart phones or laptop computers to access the web have experienced directly how this kind of computer networking activity has revolutionized human life on technical levels (e.g. applets) as expected, but also on social levels (for example, the websites Facebook and Twitter).

The following short account of web usage is based on Chap. 13 in the book "Java, an Introduction to Computer Science and Programming-second edition" by Walter Savitch, Prentice Hall 2001, where the reader can find further details. We begin with a description of HTML.

HTML

The HT stands for "Hypertext", which connotes a higher form of text that provides codemarks which are used to edit or "markup" (the M) ordinary text, for example, codemarks to specify headings, paragraph beginnings, etc. in ordinary text as would be marked by a traditional copy editor of the webpage document. But there are also some unusual special codemarks with associated text that cause connections or *links* to be made to another webpage when activated by clicking on the associated text with a mouse. (See below.) HTML statements are in the form of commands having a well-defined syntax. A web browser can read an HTML command by parsing the syntactic structure. Most HTML commands have the following structure:

< command _ name> text </command_ name> .

For example, to make the text "My Home Page" a largest heading the HTML command is as follows:

<H1> My Home Page </H1>

For a smaller heading the command name is H2 and so on. Some statements do not require a closing name. For example,

<BR> text

denotes a *break* or new line for text and <P> text denotes a new paragraph. Note that HTML is not case sensitive.

To form page layouts to be read and displayed by a browser there are commands to the browser like <center> text </center> that cause the text to be centered on the webpage when displayed. HTML can manipulate files created with any text editor but the filename must end with. html. Explanatory comments can be inserted in an HTML document by the notation <! - -Comment text-for-comment - ->. The beginning of an HTML document is denoted by <HTML> and the end by </HTML>.

The main part of a webpage document is called the *body* and is enclosed in two markups <BODY> and </BODY>. A second optional part is the email address of the document "owner" (or maintainer) and is marked off by <ADDRESS> and </ADDRESS>. These and other markup commands specify the content and the format of the display of a webpage by a browser.

To insert a picture (e.g. a photo or other image) in an HTML document the following command is used:

<IMG SRC = "file-with-picture" >

where file-with-picture is a path to the picture file. For example, suppose the picture is in the file ~ picture.gif in the directory ~ images which is a subdirectory of the directory where the HTML page is. The following command will insert the picture in the page:

< IMG SRC = " ~images/~picture.gif">

Now, the power of the web is based on the ease with which the web can be *browsed* or "surfed", that is, the ease with which different websites can be successively accessed. This facility is provided by the HTML *link (or hyperlink*) active command mentioned above. Its syntax is as follows:

<A HREF = "path- to- document" text-to-click-on </A>

For example, a link to Savitch's home page would be given as

<A HREF = "http://www-cse.ucsd.edu/usres/savitch" Walter Savitch </A>

A link can be inserted anywhere in the document and the text -to-click-on part will be underlined in the browser display. If the person browsing the page document clicks on the underlined text, the browser will automatically access and display the webpage given by the path- to- document part (the URL). In this example, a click on the underlined text Walter Savitch in the browser display window will activate access to and local display of Savitch's remote home page at UCSD in La Jolla California. This action is implemented by the program HTTP, which determines an Internet network transmission path from the current website node to the website node specified by the URL in the link and automatically (without further user action) accesses the latter, displaying its webpage.

Applets

Applets are relatively small Java programs that can be embedded in an HTML document for general use. To give even a simple example of an applet embedded in an HTML document requires that the reader knows some details about the Java language or about C++. In particular, the reader must understand the basic ideas of the concept of *classes* in these languages. We refer to the above- named book by Savitch for definitions of classes. Also see Chap. 4 above. Here we shall simplify and just say that a *class* is what was formerly called a *data type*, that is, it consists of elements called *objects* and operations (called *methods)* that can be performed on the objects. For example, the built-in class int consists of the integers and includes the familiar methods (operations) add and multiply. Classes can be defined by the programmer in very general terms. Also, a class to be defined can refer to another one that is already defined and be a descendant of it, availing itself of all the methods already defined in the parent class, a powerful way to build up complex classes.

From the Savitch book we borrow the following example of an applet as a class which defines an old-style adder machine which is used to add up a column of numbers. The adder is programmed by a GUI program as a class to be displayed by the web document in which it is embedded in the user-friendly shape of a box or "panel" looking like an actual adder machine and consisting of (1) an inputoutput "field" for inputting user numbers, (2) a small area shaped as a user "button" for "add" to activate the additions of the succession of numbers entered in the inputoutput field and (3) another "button" to "reset" the sum to zero. The GUI panel is formed by the HTML document which is being viewed by the user. Most of the Java applet consists of Java formatting statements to shape the GUI panel and define its user buttons. (See the Savitch book.) We shall omit these Java statements and give only the part of the applet which specifies the addition of the numbers, which must be converted from strings to numbers and back.

First, there is an applet class declaration statement such as public class Adderapplet extends Japplet

This is followed by statements defining the GUI panel and then by the following statements defining the main operations (methods) of the class:

```
            public void  Actionperformed(Actionevent e )
   (if (e.getActioncommand(). equals ("Add" )
     then    ( sum = sum + stringtodouble(inputoutputfield.gettext()  ;//add
  converted double-precision input field;
         inputoutputfield.settext(double.tostring(sum)) ; //convert back to
         string format;
   else if e.getActioncommand().equals ("reset")

  (sum =  inputoutputfield.settext"0.0")
```

We have made use of the built-in class operation *get* and the class dot notation for applying class functions (the e in this example).

Now to place this applet in an HTML document we use the HTML command

```
<APPLET CODE = "AdderApplet.class " WIDTH = 400 HEIGHT = 200>
</APPLET>
```

A web user can access and use this adder applet by browsing to the web page in which it is embedded. The user then inserts numbers in the input field and presses the add button by clicking on it (like an ordinary physical adding machine). Although elementary in concept, this example illustrates the power of web resources.

References

NAUR, Peter (ed.), "Revised Report on the Algorithmic Language ALGOL 60." *Communications of the ACM*, Vol. 3 No.5, pp. 299–314, May 1960.

JENSEN, Kathleen, WIRTH, Niklaus, "PASCAL user manual and report", = *Lecture notes in computer science* ; vol. 18., Berlin [etc.] : = Springer, 1974., 1974. = 20

S.C. Johnson, " Yacc: Yet Another Compiler Compiler", Computer Science Technical Report #32, Bell Laboratories, Murray Hill, NJ, 1975.

WIRTH, Niklaus., *Programming in Modula-2*, Berlin, = Heidelberg Springer, 1982.

M. Marcotty & H. Ledgard, *The World of Programming Languages*, Springer-Verlag, Berlin 1986., pages 41 and following. Th. Estier *, CUI - University of Geneva*

Computer Organization and Design by David Patterson and John Hennessy, Morgan Kaufman 2005.

Part II

Chapter 5
The Hardware Side

Edward K. Blum

Chapter 2, which traced a brief history of computation, included a short survey of early *computer hardware* devices invented by Leibniz, Pascal and Babbage. These early brave attempts to expedite computation by means of machines were hampered by their forced reliance on the only available technology of their eras, the mechanics of gears and wheels. We shall regard the Pascal and Leibniz machines as interesting museum pieces. Only Babbage's nineteenth century computer, called the *analytical engine*, had design features, such as programmability and punched card input of data and programs, that influenced the design of modern computers. It was not until the twentieth century that electric/electronic components, such as electromagnetic relays and electronic vacuum tubes and transistors, were widely used in the fabrication of *computer hardware*. We shall begin our exposition of the *Hardware* Side of Computer Science with these electric/electronic–based devices. Our point of view of *Computer Hardware*, as explained previously, is the viewpoint apparently adopted by Turing which we characterize as viewing Hardware and Software as two indissoluble related sides of the "computation coin". He emphasized the Software side in his initial studies as possessing the motivating ideas for a theoretical computer design but was inevitably drawn into detailed engineering studies on the Hardware side as a result of his involvement in the ACE computer project.

The ACE acronym stands for *Automatic Computing Engine*, the word "engine" used in deference to Babbage's analytic engine perhaps and the word "automatic" used as a fashionable term for new machines of that era to indicate that their computations proceeded under "self-control" without constant external human intervention or supervision as with desktop machines. The ACE project had as its goal the design and fabrication of a practical hardware computer based on theoretical ideas conceived primarily by Turing. It was for several years a pet project of

E.K. Blum (✉)
Department of Mathematics, University of Southern California, Los Angeles, CA, USA
e-mail: blum@usc.edu

E.K. Blum and A.V. Aho (eds.), *Computer Science: The Hardware, Software and Heart of It*, DOI 10.1007/978-1-4614-1168-0_5,

Turing's, as recounted in some detail in the biography of Turing, "Alan Turing the Enigma" by Andrew Hodges (1983, Simon and Schuster). Turing, a mathematician we and many others (e.g. the originators of the Turing prize) have anointed as the father of modern Computer Science, was gifted in many fields of mathematics including the wartime-crucial one of cryptanalysis for which specialty he was early employed in the British Code and Cypher School during World War 2. Besides his mathematical theorizing, he was the inventor of a secret British hardware device that helped to break the German Enigma coding scheme, thereby significantly helping to defeat the German navy's submarine assaults. Like other mathematicians we have mentioned in our history of computation (e.g. Leibniz, Pascal, Babbage and von Neumann) Turing was not a one-sided ivory-tower mathematician but rather was conversant with many aspects of the practical Hardware side of computation. As a consequence of his experiences, direct and indirect, with cryptographic decoding machines during the war, such as the Colossus with its 1,500 vacuum tubes, he learned a significant amount of pertinent electronic engineering and practical physics. His experiences were not merely theoretical but involved physical experiments in his home, which was often cluttered with experimental hardware apparatus items.

After the war, around 1945, he was appointed to a position at the National Physical Laboratory, which had a mathematics section involved in developing large-scale computers for the British government. His position allowed him the time and assigned him the objective to develop ideas for a practical computer. Turing's long-standing and governing idea for a practical computer was ambitious but rather less practical than the ideas many computer engineers were then considering. Turing's grand objective was to build a machine that would implement his original concept, propounded in his 1936 paper, of a **universal Turing machine**, in particular, its central idea of a stored-program computer (See Chap. 3). Turing began a lengthy report on a proposed version of a universal machine which was dubbed the ACE. Meanwhile, with the availability of the new electrical technology of reliable relays and vacuum tubes, others had taken up the task of building real computers, among them the German engineer K. Zuse working with relays, an American engineer G. Stibitz at Bell Laboratories also working with relays, the physicist H. Aiken at Harvard University also working initially with relays and finally the so-called "wizard" von Neumann working as an adviser on the ENIAC computer project at the University of Pennsylvania. The ENIAC was an electronic computer designed by the electrical engineers J.P. Eckert and J. Mauchly. The ENIAC (Electronic Numerical Integrator and Calculator) was funded by the military as a special-purpose computer intended for calculating artillery range tables. Its initial versions contained about 19,000 vacuum tubes and demonstrated that such a large assembly of tubes could be designed so as to operate reliably despite many unreliable components.

Now, to understand Turing's approach to the design of the ACE, it is useful to recall (Chap. 3) the two main principles and parts in the operation of a Turing machine M: first, its tape as a "memory" for storing data (and also programs of instructions considered as data in the case of a universal machine U) and second, the finite-state control C of M which determines the sequence of execution steps of M. A *program* M(I), which is written as a list of instructions which define the computations performed by machine M on various tape data inputs, can be stored in a recognizable format on the tape of a universal machine U. The finite-state control of U can then read the stored-program M(I) and cause U to execute the sequence of execution steps of M as specified by the instructions in M(I). The result is a *sequence of steps*, Seq(M), in which specified operations are performed on the tape contents to effect the computation which M is designed to do. The sequence Seq(M) is, in fact, the specified computation done by M and must be determined from the program M(I) by having U *simulate* the machine M's Control process. As shown in Chap. 3, the simulation process which derives the sequence Seq(M) from the program M(I) can be a quite lengthy computation process. How to build a hardware machine U which can do this? This was the design problem which Turing addressed.

Recall that for a Turing machine, the *stored-program* M(I) is a textual list of quadruple instructions. For a machine of a more practical kind, a *stored-program* M (I) is a list of statements such as the assignment statements and conditional statements discussed in the preceding Software chapter. Turing's *programming construct* in his machines M for deriving a sequence of steps from a stored-program M(I) utilizes the concept of an internal *present state* q of control C of M combined with the scanned character x of M's tape and a *next state* q#, all represented in a simple instruction quadruple qxOPq#, where OP is an operation to be performed by M. The program M(I) is a sequence (I(0), I(1), . . ., I(n)) represented as a list of such instructions I(j), but Seq(M) does not necessarily follow the ordering in this list. The interpretation of q as an instruction label or memory *address* (implemented by a *program counter* say) of a stored instruction permits the *present* and *next* execution steps to be determined by any instruction in M(I). In fact, we see that Turing's programming construct qxOPq# allows an implementation or interpretation by means of a stored software conditional statement stored at address q as follows

q**:** **If** x then OPq# **else** . . . .

which we described in the Software chapter. The **If** clause applies if the present Control state is q and the scanned character is x and causes operation OP to be executed and then a transfer to a statement having the address q#. The **else** clause applies if the scanned character is not x, but another character, y say, and its action is given by a quadruple in M(I) of the form qyOP1q##, which causes OP1 to be executed and a *state transition* to q##. This is the alternate branch of the execution path. Again, the state transition is interpreted as a transfer of control to a quadruple stored at address q##. This implementation illustrates our contention that the Software side often dictates the Hardware side of a computer. Turing's stored-program concept provides the means for U to perform the sequential control of M as specified by the program M(I). It applies to machines with other programming statement formats in a program M(I), as long as M(I) allows for the inclusion of a conditional statement. Recall that the C++ and Fortran programming languages provide conditional statements of the form **if**. . . **then**. . .**else**.. The Babbage analytic engine programs allowed such conditional statements. However, the Zuse, Aiken and Stibitz machine programs did not, whereas a later general-purpose version of the ENIAC called EDVAC (Electronic Discrete Variable Automatic Computer) followed a von Neumann design which included conditional statements in programs stored in a machine *memory* module. Statements for EDVAC could be accessed, processed and executed like the quadruples in a universal Turing machine as just described. i.e. EDVAC worked like a stored-program computer carrying out an execution sequence determined by instructions stored at memory addresses. Von Neumann then set up shop at the Institute for Advanced Study (IAS) in Princeton in collaboration with the mathematician H.H. Goldstine and the logician Arthur Burks, issuing their famous 1946 US Army Ordnance Department report, "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument". This report propounded most of the *architectural* (i.e. organizational and functional) concepts found in modern computers of the 1940–1960s such as a single main *memory module*, a *central processing unit* (CPU) that had priority access to the memory, and *input and output* units (IO) and led to versions of an IAS computer built by the engineer J. Bigelow which became known first as the MANIAC and then the JOHNIAC in tribute to its wizard designer. The US Army report was freely and widely circulated in the USA and Great Britain, including Cambridge where it served as a guide to M. V. Wilkes in building the EDSAC (Electronic Delay Storage Automatic Calculator) computer.

Although Turing and von Neumann were then on the same hardware design page, namely, using the stored-program computer design concept in which both data and programs were stored in a memory module of a (universal) machine, they faced a major problem, namely the size and structure of the memory module which had to allow access to a potentially large store of instructions at electronic speeds. von Neumann's design solution, with a centralized memory module for both data and programs, referred to as the *von Neumann architecture*, set up a paradigm for many modern computers but was criticized as causing a bottleneck that slowed execution (See the Backus Turing Award lecture reference in Chap. 4. The memory bottleneck was eventually alleviated by adding auxiliary quick-access memory hardware components for the stored programs).

At this juncture, with the acceptance of the von Neumann architecture, the Computer Hardware development task became a battle focused on an effort to develop fast large memory. In Great Britain, this battle was fought rather fiercely by two competing universities, Cambridge and Manchester. Essentially there were then two competing technologies for constructing memories: one called *circulating pulse delay lines* (a partly mechanical device in which data was stored as sound pulses circulated in mercury delay lines) and the other in which data was stored as continually refreshed electrostatic charges on cathode ray tubes (CRTs), a completely electronic technique. Turing at first experimented with circulating pulses in delay lines but then moved to a position at Manchester, where the CRT approach was favored by the Manchester engineer F. Williams. The CRT memory was also adopted in the USA by RCA and later by IBM. It was successful but its lifetime was short, being displaced by the large magnetic core memory developed at MIT.

Returning to the stored-program concept and the problem of specifying and determining a computation's sequence of steps that a stored-program machine M can execute, Turing's notion of a finite-state control C and the simple technique of specifying operation steps by quadruples of the form qxOPq# leads in fact to a practical general hardware scheme for deriving a computation sequence for any modern computing machine M. Since many different programs can be stored in the memory module of M and executed like the programs stored on the tape of a universal Turing machine, we can regard M itself as an approximately universal machine. Control of execution of any program stored in M's memory can be done in a manner similar to the Control of execution of a program stored on the tape of a universal machine, using a device like Turing's instruction addresses q(0), q(1), ..., say implemented by an electronic *program counter*, as explained above.

The finite-state Control and the stored-program concept have been adopted as the basis of execution control of a type of machine known as a *MIPS machine*, which is described at length in the book "Computer Organization and Design" by D. A. Patterson and J.L. Hennessy, 2005, Morgan Kaufmann Publishers. Actually, MIPS was a real *microprocessor* built in 1984 as an *integrated circuit* (see Appendix on logical circuitry below) on a silicon *chip* which was 1 cm square. It had a *Clock* (see below) running at 20 MHZ. Many modern computers M are closely related to those of the MIPS type in that programs written in high-level languages like C++ and JAVA and assumed to be executable on M can be compiled into machine-level MIPS programs consisting of basic MIPS assembly language computation statements of the general forms as follows:

X = A + B or X = A*B etc. specifying operations on data in memory locations A and B;

memory load statements like lw $t0 x which loads a data word from memory location x into a *special machine register* $t0;

sw $t1 x which *stores* (i.e. writes) a word from register $t1 into memory location x.

These statements can be easily implemented by standard Boolean logic combinational circuits on the MIPS chip. Thus MIPS serves as a *prototype* hardware computer. Compiled MIPS programs look very much like simple assembly language programs at a near-machine-level in many machines M (See Software Chap. 4). As stated in the Software chapter, the implicit assumption that most Fortran, C++ or JAVA programs will run on a contemplated machine M in effect imposes many hardware design specifications on M before it is built. Hardware designers make this implicit assumption, since otherwise their machine M would be ignored by potential users. Thus, by this MIPS approach to hardware design, software (i.e. computation) requirements usually precede the hardware design of a computer.

Let us therefore assume that compilation into MIPS assembly-level programs can be done for a machine M's various high-level programs. If, for example, C++ is the programming language for M, then one assumes that M can execute all or most C++ statements. It can actually do this if there is a compiler from C++ into an assembly language consisting of the above MIPS format statements which defines much of the hardware of M. Thus, hardware design becomes a matter of compilation of high-level programs into some assembly language which defines M. Now, rather than use an assembly language for M, we can use the above assembly language for MIPS if we further assume that M is a MIPS type machine, that is, assume that the assembly language for MIPS can be translated into an assembly language for M and conversely. In this sense, M is equivalent to a MIPS type machine. Of course, in practice, we allow the designer of M to specify various special properties of this translation from M to MIPS which hold only for this particular M. Therefore, the hardware designer assumes that for any program P to be run on M the compiler produces a MIPS style assembly program. Let us designate this compiled MIPS program by MIPS(P). To design a hardware version of M to execute all programs P it suffices to design a hardware version of M to execute all MIPS(P) compilations. In this way, MIPS provides a general hardware design vehicle for a large class of machines M. This general MIPS design method is detailed in the Patterson-Hennessy (P-H) book cited earlier. We do not have space to cover all the MIPS(P) programs as is done in P-H, but we can illustrate the method by a few key examples. We keep in mind the theoretical objective that an arbitrary Turing machine TM corresponds to a program TM(I) that must be executed by the universal ACE Turing machine. Here, we have instead an arbitrary program P, or rather a compiled assembly program MIPS(P), that must be executed by a MIPS type (approximately) universal machine M to be designed.

We begin by setting up a finite-state Control C for machine M like that in a Turing machine. The internal states q of C are replaced by memory addresses of the MIPS statements in MIPS(P). We can simply use integers for the q's, starting with q = 0 for the first statement in the list MIPS(P) and proceeding through the values q = 4, 8,. . .given by the start positions of the statements in the sequence MIPS(P), assuming, as in P-H, that each basic MIPS statement occupies 4 bytes of memory (*A byte* is a sequence of 8 bits which encodes integers, alphabetic letters and some punctuation symbols. A *bit* or binary digit is a 0 or 1. A *word* in computer memory often consists of 4 bytes that is, 32 bits). The value of q for the *current statement* to be executed at any time tn is to be held in a MIPS *register* called the Program Counter (PC). A *register* is a type of memory unit that can be fabricated, for example, from standard electronic logic gates connected to form flip-flops (See Chap. 3 Appendix G, Boolean algebra, and the Appendix on Logic Circuitry which follows below). By straightforward memory addressing circuits, designed as Boolean switching functions, a register's data contents can be easily connected to and written into any memory location x of M by the MIPS sw x operation, as stated earlier, or data can be loaded from any memory location x into a register by the MIPS lw x operation. The sequence of steps for a computation is determined by proceeding in sequence through the list of stored compiled MIPS statements in MIPS(P), executing them at successive times t(n), n = 0, 1,. . .and incrementing PC by 4 after each execution until a conditional statement is encountered. MIPS programs contain such conditional statements just as we have seen that the Turing quadruples do in their interpretation by the finite-state control. A conditional statement, when the **else** clause applies, is implemented by resetting the program counter PC to a value specified in the **else** branch. This implements the sequencing of the steps in a computation of MIPS(P). Although straightforward, the implementation of a conditional statement is what allows program execution sequencing to be quite general and more than a trivial exercise in automatic text sequencing. Thus we require that MIPS, and therefore M, possess the branching ability afforded by a program counter.

Now, a few moments thought will suggest that something is still missing from the implementation of Control. If the above process of implementing the individual steps in a computation sequence is to actually run as a physical process on computer M, then after M completes the nth statement's computation step at time t(n), there must be an explicit *Next signal* to further activate M to *fetch* the next (n + 1)st statement specified in MIPS(P) at time t(n + 1). Otherwise, M could simply rest inactive and wait idly after completing step n. All along this activating Next signal has been an implicit feature of Control, implicit in the fundamental dynamics of its postulated discrete-time sequence of states q(0), q(1), . . .and execution steps. By contrast, in a classical physical continuum-time dynamical system, for example a vibrating spring, its state S depends on a time variable t which is a real number and after the *present state* S(t) is set up at time t, there is no N*ext state* since there is no "next" time following t (*Successor* states at times beyond t are determined by temporal dynamics structures such as differential equations and other continuum-based mechanisms). For MIPS computers like M and Turing machines, we now make the discreteness of time explicit by introducing a discrete *Clock* signal as part of the hardware. This defines the N*ext-signal step* concept. Every Turing machine and MIPS computer M has a discrete-time Clock which emits a periodic two-level Clock signal as a sequence of electric pulses, say at levels 0 and 1, at discrete times t(n). The period interval is called a *Clock cycle*. These pulses govern the state transitions and computation step sequencing. In a real hardware computer designed along the lines of Turing's finite-state discrete-time control as just outlined. Each hardware subprocess (e.g. one of the three MIPS basic processes mentioned earlier) used to execute the *next* (n + 1)st step at time t(n + 1) in the computation sequence for MIPS(P) must be initiated by a Clock pulse so that the subprocess occurs only after the *curren*t nth step is completed in the current nth Clock cycle. This not only keeps the computation running (until explicitly halted), but it also eliminates possible ambiguity as to the *value* in a memory cell or register in any Clock cycle. The value in the (n + 1)st cycle step is that which exists after the nth cycle step is completed. If reading and writing data values were not separated into different Clock cycles there could be ambiguity as to the values read. Thus, Clock pulses must be part of the input to the various combinational circuits which define Turing machine operations and the operations specified by MIPS statements (See the Appendix below on Logical Circuits). These circuits with clock pulse inputs become *sequential circuits* i.e. their outputs at time t(n + 1) depend on the state value at time t(n) and any new inputs at time t(n + 1).

In the P-H book, a Clock is introduced to generate cycles of control steps at discrete times t(n). The situation is a bit more subtle in practice since the real physical circuit switches and gates described in the Appendix below do in fact operate as circuits in continuum time t. Therefore, there is a problem as to exactly when to input a clock pulse to a logic element. The pulse has some width and can be input at any point in a clock cycle. In the P-H main text, this problem is set aside in favor of an engineering *Clocking Methodology*, called *edge-triggered*, which dictates that a value in a logic element is triggered (updated) by a clock pulse only at the instant of the edge of a clock pulse. This works as a practical engineering design methodology provided that we further assume, as P-H does, that a clock cycle is long enough so that the values in logic circuit elements have been stabilized when the next edge trigger is supposed to occur. The timing of stabilization of circuit waveforms is classical electric circuit theory involving resistors, capacitors and inductances and is presented, for example, in the book, “Foundations of Analog and Digital Electronic Circuits”, by A, Agarwal and J. H. Lang, 2005, Morgan Kaufmann. The logical designer of the Boolean functions which perform the operations in the successive Clock cycles in a machine M must choose the Clock cycle width to be compatible with the time constants in the classical circuit properties of the gates that make up the Boolean functions. We shall assume, along with P-H, that this can be done by practical engineering techniques based on known circuit time constants for standard circuit devices implemented on *integrated circuit chips* (See Appendix below). A more detailed design would involve such time constants as those in the charging and discharging of capacitive circuit components (See, for example, the book Mathematics of Physics and Engineering by E.K. Blum and S. Lototski. World Scientific Press 2006).

In P-H, a detailed prescription of the design of the control C is given for each basic MIPS statement. Here, we shall illustrate the method with two examples. This will indicate how the entire design can be done for a MIPS computer in terms of elementary MIPS machine-level operations. Before proceeding with these examples, we shall digress briefly to point out that there are now computer chips which are not single MIPS computers but rather are composed of multiple MIPS processors which communicate with each other in concurrent computations in order to speed up the solution of a problem. These multi-processor systems have multiple Controls and clocks. For large multi-processor systems, one protocol for the design of such systems involves a message-passing standard like MPI which is discussed in Chap. 7 on high-performance cluster computing.

As the first MIPS example, we consider a simple arithmetic assignment statement in program P,

$$A = B + C,$$

as produced by a compiler. In a MIPS machine, arithmetic operands must be in special hardware locations called *registers*. This simplifies the machine design and speeds up execution of arithmetic by having fast access registers. Likewise, data paths are executed with registers. Assume that the contents of cell C has been written into register $t1 and that of B into register $s1. Assume further that cell A will be connected to register $t1. The above assignment statement is compiled further into the following three MIPS statements:

lw $t0 , $t1 // the value in C is loaded into $t0
add $t0, $s1, $t0 // $t0 gets the value of B + C
sw $t0 $t1 // value B + C is written into $t1 for subsequent loading into A

To execute a MIPS statement like **add** $t1, $t2, $t3 in the above list, the Control in P-H uses four steps (all within one clock cycle) as follows:

1. The instruction is fetched and the PC incremented as explained earlier.
2. The registers $t2 and $t3 are read and the Control sets control line values for succeeding steps.
3. The arithmetic logic unit (ALU) of the machine performs the add indicated.

4. The resulting sum is written into register \$t1.

As a second example, we consider the MIPS instruction

lw \$t1, **offset**(\$t2)

which operates on addresses of data to be written. In P-H, this is implemented in five MIPS steps as follow:

1. The instruction is fetched from memory and the PC incremented.
2. Register \$t2 is read.
3. The lower 16 bits of the instruction (the **offset**) is added to \$t2 in the ALU.
4. The sum in the ALU is the address of the data memory.
5. The data from this memory is written into register \$t1.

These examples illustrate how machine M is designed as a MIPS type of machine which uses registers and an arithmetic logic unit (ALU) to carry out machine-level operations to execute any statement in a high-level language program like Fortran, C++ or JAVA. Such statements must first be compiled into MIPS assembly language statements. For compound arithmetic expressions like (B + C)*D in a high-level program statement the compiler might construct an *evaluation stack* of operations and operands, such as + BC*D, which first causes the evaluation (B + C) and then is followed by multiplication of the resulting sum by D. The writing of compilers is a special art fraught with skills that we do not have space to cover.

Besides the compiler program which can be utilized by machine M when reading a stored high-level program MIPS(P) to be executed there are other "services" provided by an *operating system* (Chap. 6) in M. These include the *loading* H of standard subroutines needed by MIPS(P) and *linking* them to MIPS(P).

Appendix 1

Logic Circuits

E. K. Blum

The general organization of a computer's hardware system is called its architecture. For example, we have mentioned the von Neumann architecture. The architecture of the MIPS computer was outlined in Chap. 5 above, which designated the MIPS major parts as memory, a Control unit and processor, and input–output units. On a more detailed level, we further designated the existence of MIPS operations, such as arithmetic operations, by invoking them in various program units in the MIPS assembly language produced by a compiler. The full compiler results were not specified, since this is a major job in itself. Rather it was assumed reasonably that basic program units (such as primitive statements like C = A + B) can be extracted from the compiler's assembly language results. Such primitive statements constitute a fairly high-level software specification of the MIPS low-level hardware parts, as we illustrated in Chap. 5. As a physical machine, MIPS was originally fabricated as an integrated circuit (IC), called a chip, in 1984 having the basic parts designated by the primitive compiler statements (e.g. registers to load data to/from memory, an arithmetic logic unit (ALU) capable of executing the designated operations like +, •, etc.) The design and fabrication of the ALU is a separate task that has been done in many computers in many ways. We take the ALU as a given part of the MIPS hardware.

The MIPS chip was 1 cm square and had a clock running at 20 MHZ. It consisted of *logic circuits* built on a silicon substrate. In this Appendix, we describe the main constituents of these logic circuits. They are the physical MIPS hardware at its lowest level. Similar chips are the real hardware of other computers such as "Intel-inside" personal computers.

The physical hardware components of a computer architecture such as that of MIPS are fabricated as Boolean combinational circuits (described mathematically by Boolean algebra formulas as in Chap. 3 Appendix G) or as Boolean sequential circuits with a clock input to actuate the execution Control process. These two types of logic circuitry are built from electronic Boolean logic gates (e.g. AND, OR, NOT, NAND, NOR etc.) and *electronic switches*. The gates and switches are fabricated as parts of a connected *integrated* circuit (IC) and are assembled by lithography and other processes on a *silicon chip*. The key silicon component fabricated on a chip is the semiconductor called a *transistor* created from silicon crystals by a process called *doping*, which adds impurities to the silicon. Other chip circuit components are also created by doping, that is, by adding various other elements to very small parts of a layer of silicon so that the small parts then function either as lumped *conductors* or *insulators* in one connected *integrated circuit*. In this Appendix the reader is introduced to various types of transistors and some of the silicon-based circuits they operate in. The reader is advised to merely scan these low-level hardware details just to get an appreciation of how much this technology has evolved from "silicon valley". The technology of IC manufacture is a magnificent engineering achievement by several companies such as, for example, Texas Instruments, Analog Devices, Intel, AMD, IBM, Samsung and many others. The fabrication process that does the "doping" is rather intricate, involving a *photo-etching* procedure that imprints patterns on a silicon surface and then dissolves the parts not imprinted. *Masks* are used to shape the IC patterns and fabrication may entail several layers of masks. Parts of the unmasked areas are later filled in with conductors of copper or aluminum to form an electric circuit. Further details of IC fabrication are beyond the scope of this book and furthermore are often proprietary.

A *transistor* is an arrangement of *semiconductor* solid material having an electrical conductivity, at room temperature, between that of a conductor and an insulator. Hence the name. Materials most commonly used are silicon, gallium-arsenide, and germanium, into which impurities have been introduced, as stated above, by a process called *doping*. In *n-type* semiconductors the impurities or *dopants* result in an excess of electrons (negative charges); in *p-type* semiconductors the dopants lead to a deficiency of electrons and therefore an excess of positive charge carriers called "holes."

Although there may be some question regarding priority, the invention of the transistor is usually credited to physicists John Bardeen and Walter Brattain. at ATT's Bell Labs in 1947. They observed that the output voltage of a germanium crystal in a circuit can be much greater than the input voltage. In 1954. at ATT's Bell Labs, Solid State Physics Group leader William Shockley saw the potential in this physical behavior, and worked to develop it into the *junction transistor*. He is viewed as the "father of the transistor". The name 'transistor' is shorthand for the term "transfer resistor", which refers to the resistors used in a classical *lumped circuit* engineering representation of the input–output *equivalent* circuits of a transistor (See figures below).

Field Effect Transistor (FET)

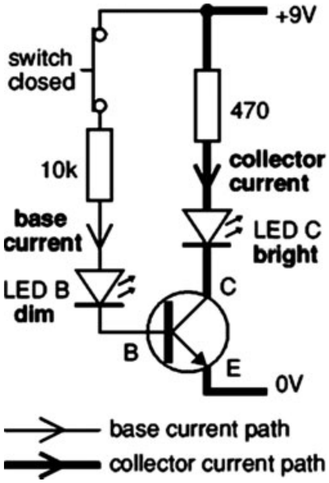
An important type of transistor developed after the original junction transistor is the field-effect transistor (FET). It draws virtually no power from an input signal, overcoming a major disadvantage of the junction transistor. An *n-channel FET* provides a conducting path or *channel* mainly of *n*-type silicon material that is built as two separated n-type regions on a substrate of p-type silicon. This is called an n-p-n configuration of the channel. The two n-type regions are separated by a p-type region. Two conductor terminals attached to the two n-type regions of the channel are called the *source* and the *drain* to indicate the direction of intended channel current flow from one n-type region to the other across the p-type gap. To control channel current flow, the p-type gap is overlayed with a thin insulating layer of silicon dioxide on top of which is affixed a conducting polysilicon layer called a *gate*, which serves as a third terminal. A voltage applied to the gate terminal creates an opposing electric field in the gap directed so that zero or little current flows across the gap formed by the n-p-n configuration. For this reason it is called a *reverse voltage*. Variations of the magnitude of the reverse voltage cause variations in the resistance of the total n-p-n channel, enabling the reverse voltage to control the current through the channel that would be produced by a voltage applied across the source and drain terminals. The channel current can be made to vary from near zero to a full value, as with an off-on switch. A *p*-n-p configuration works the same way but with all polarities reversed.

The first silicon transistor was produced by Texas Instruments in 1954. The first MOSFET type of transistor (see below) actually built was at Bell Labs in 1960. The transistor is the key active component in practically all modern electronics circuits and is considered to be one of the greatest inventions of the twentieth century. Its importance derives from its many circuit functions and its ease to be mass produced by an automated process that achieves very low transistor costs. Although several companies each produce over a billion individually packaged (*discrete*) transistors every year, the majority of transistors now produced lie within integrated circuits (*chips*) along with *diodes, resistors, capacitors* and other electrical circuit *lumped* componentshttp://en.wikipedia.org/wiki/Electronic_ components, so as to produce complete electronic circuits. A Boolean logic gate (AND, OR etc.) can consist of up to about 20 transistors whereas an advanced microprocessor chip, as of 2009, can include as many as 2.3 billion transistors.

The Metal Oxide Semiconductor Field-Effect Transistor (MOSFET)

The metal-oxide semiconductor field-effect transistor (MOSFET) is a variant of the FET (with source, drain and gate terminals) in which the gate terminal is separated from the main transistor n-p-n *output* channel by a layer of metal oxide, which acts as an insulator, or dielectric. The electric field produced by a voltage applied to the gate extends through the dielectric and controls the resistance of the channel between source and drain ends. In this device, the *input* signal, which is applied to the gate, can, depending on its polarity, increase the current through the channel or decrease it. As cited above, the invention of the transistor is usually attributed to the American physicists John Bardeen, Walter H. Brattain, and William Shockley, later jointly awarded a Nobel Prize. It was announced by the Bell Telephone Laboratories in 1948; it was also independently developed nearly simultaneously by Herbert Mataré and Heinrich Welker, German physicists working at the Westinghouse Laboratory in Paris. Since then, many types of transistors have been designed. At one time, only discrete (single) devices existed; they were usually sealed in ceramic, with a wire extending from each terminal (source, drain and gate) to the outside, where it could be connected to an electric circuit. As remarked above, although discrete transistors are still used, the majority of transistors are now built as parts of an integrated circuit *chip*. Transistors are used in virtually all electronic devices.

The *n-p-n* junction transistor is similar to the FET. It consists of two *n*-type semiconductors (called, as one might expect, the *emitter E* and *collector C*) separated by a thin layer of *p*-type semiconductor (called the *base B*). The transistor action is such that if the electric potentials on the segments E, B and C are properly determined, a small (**input**) current between the base and emitter connections results in a large (**output**) current between the emitter and collector connections, thus producing current *amplification*. Other circuits are designed to use the transistor as a switching device; current across the base-emitter junction creates a low-resistance path between the collector and emitter resulting in a closed switch connection. The *p-n-p* junction transistor, consisting of a thin layer of *n*-type semiconductor lying between two *p*-type semiconductors, works in the same manner, except that all polarities are reversed. Most transistors used today are of the n-p-n configuration because this is the easiest type to make from silicon. Shown here is a schematic circuit diagram downloaded from The Electronics Club web page. The diagram shows the two current paths through a transistor. This circuit can be built with two standard 5 mm red LEDs (light-emitting diodes) and any general purpose low power n-p-n transistor (BC108, BC182 or BC548 for example). When the main circuit switch is closed a small current flows **into** the base (B) of the transistor and controls the output current. It is just enough to make LED B glow dimly. The transistor amplifies this small current to allow a larger current to flow between its collector (C) and emitter (E). This collector current is large enough to make LED C light brightly. When the circuit switch is opened no base current flows, so the transistor switches off the collector current. Both LEDs are off. This arrangement where the emitter (E) is in both the controlling circuit (base current) and in the controlled circuit (collector current) is called *common emitter* mode. It is the most widely used arrangement for transistors.

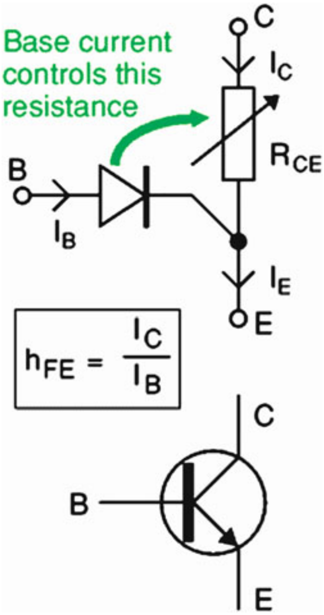


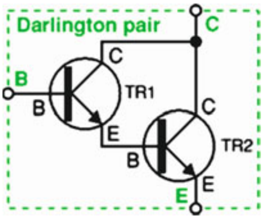
Functional Model of an NPN Transistor

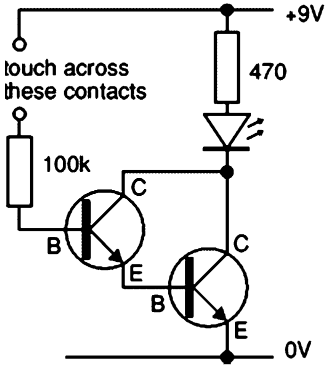
The physics model of operation of a transistor is difficult to explain and understand in terms of its internal atomic crystal structure. It is more helpful to use an electrical functional model given in the Electronics Club web page as shown here:

- The *base-emitter junction* behaves like a diode.
- A base current I_B flows only when the voltage V_{BE} across the base-emitter junction is 0.7 V or more.
- The small input base current I_B controls the large output collector current Ic.
- Ic $= h_{FE} \times I_B$ (unless the transistor is full on and saturated) h_{FE} is the *current gain* (strictly the DC current gain). A typical value for h_{FE} is 100 The collector-emitter resistance R_{CE} is controlled by the base current I_B:
  - $I_B = 0$ $R_{CE} =$ infinity transistor off
  - I_B small R_{CE} reduced transistor partly on
  - I_B increased $R_{CE} = 0$ transistor full on (‘saturated’)

- A resistor is often needed in series with the base connection to limit the base current I_B and prevent the **transistor** being damaged.
- Transistors have a maximum collector current Ic rating.
- **The current gain h_{FE} can vary widely**, even for transistors of the same type!
- A **transistor** that is **full on** (with $R_{CE} = 0$) is said to be '**saturated**'.
- When a transisto**r** is saturated the collector-emitter voltage V_{CE} is reduced to almost 0 V.
- When a transistor is saturated the collector current Ic is determined by the supply voltage and the external resistance in the collector circuit, not by the **transistor**'s current gain. As a result the ratio Ic/I_B for a saturated **transistor** is less than the current gain h_{FE}.
- The emitter current $I_E = Ic + I_B$, but Ic is much larger than I_B, so roughly $I_E = Ic$.







Touch switch circuit

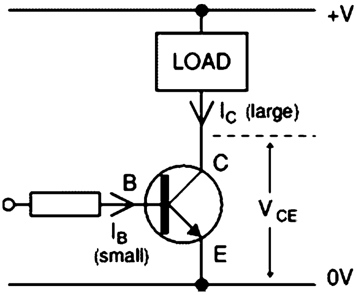
Darlington Pair Circuit

As previously remarked, transistors can be used in amplifier circuits. The amplification produced by a single transistor is not very high. To achieve high amplification we can use a Darlington pair circuit as shown. The circles with three inside segments denote transistors in which B is the base, C the collector and E the emitter. The pair behaves like a single transistor with a very high current gain. It has three external green leads (B, C and E) which are equivalent to the base, collector and emitter leads of a standard individual transistor. To turn on there must be 0.7 V across both of the base-emitter junctions which are connected in series inside the Darlington pair, therefore it requires 1.4 V to turn on.

A Darlington pair is sufficiently sensitive to respond to the small current passed by human skin and it can be used to make a **touch-switch** as shown in the diagram. For this circuit which just lights an LED the two transistors can be any general purpose low power transistors. The 100 kΩ resistor protects the transistors if the contacts are linked with a piece of wire.

Using a Transistor as a Switch

When a **transistor** is used as a switch it must be either **OFF** or **fully ON**. In the fully ON state the voltage V_{CE} across the **transistor** is almost zero and the **transistor** is said to be **saturated** because it cannot pass any more collector current Ic. The output device switched by the **transistor** is usually called the 'load'.



The power developed in a switching transistor is very small:

- In the **OFF** state: power = Ic × V_{CE}, but Ic = 0, so the power is zero.
- In the **full ON** state: power = Ic × V_{CE}, but $V_{CE} = 0$ (almost), so the power is again very small.

The tutorial procedure below is taken from an Electronics Club web page and explains how to choose a suitable switching transistor.

1. The transistor's maximum collector current Ic(max) must be greater than the load current Ic.

$$\text{load current Ic} = \frac{\text{supply voltage Vs}}{\text{load resistance } R_L}$$

2. The transistor's minimum current gain h_{FE}(min) must be at least **five** times the load current Ic divided by the maximum output current from the IC.

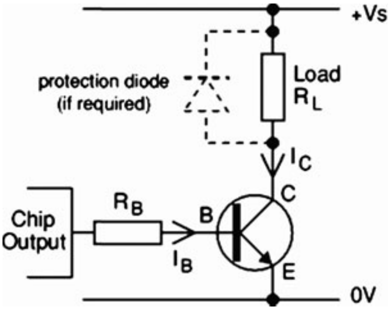
$$h_{FE}(\text{min}) > 5 \times \frac{\text{load current Ic}}{\text{max. IC current}}$$

3. Choose a transistor which meets these requirements and make a note of its properties: Ic(max) and h_{FE}(min). There is a table showing technical data for some popular transistors on the transistors page.
4. Calculate an approximate value for the base resistor:

$$R_B = \frac{\text{Vc} \times h_{FE}}{5 \times \text{Ic}}$$

where Vc = IC supply voltage (in a simple circuit with one supply this is Vs)

5. For a simple circuit where the IC and the load share the same power supply (Vc = Vs) you may prefer to use: $R_B = 0.2 \times R_L \times h_{FE}$
6. Then choose the nearest standard value for the base resistor.
7. Finally, remember that if the load is a motor or relay coil a **protection diode is required**.



NPN **transistor** switch (load is on when IC output is high)

Using units in calculations Remember to use V, A and Ωor V, mA and kΩ. For more details please see the Ohm's Law page.

Example

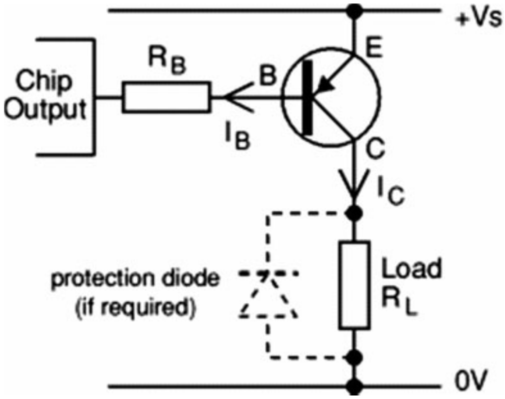
The output from a 4,000 series CMOS IC is required to operate a relay with a 100 Ω coil.

The supply voltage is 6 V for both the IC and load. The IC can supply a maximum current of 5 mA.

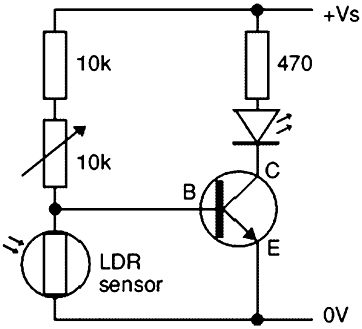
1. Load current = $Vs/R_L = 6/100 = 0.06A = 60mA$, so **transistor** must have $Ic(max) > 60mA$.
2. The maximum current from the IC is 5 mA, so **transistor** must have $h_{FE}(min) > 60(5 \times 60mA/5mA)$.
3. Choose general purpose low power **transistor** BC182 with $Ic(max) = 100mA$ $andh_{FE}(min) = 100$.
4. $R_B = 0.2 \times R_L \times h_{FE} = 0.2 \times 100 \times 100 = 2000\Omega$. so choose $R_B = 1k8$ or 2k2
5. The relay coil requires a protection diode.

Choosing a Suitable PNP Transistor

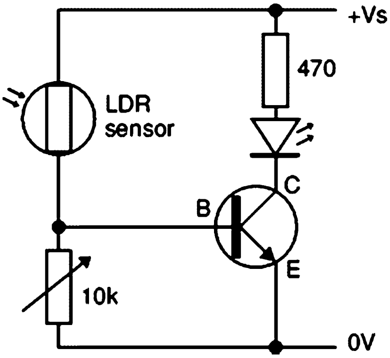
The circuit diagram shows how to connect a **PNP transistor**, this will switch on the load when the IC output is **low** (0 V). For the opposite action, with the load switched on when the IC output is **high** see the circuit for an NPN **transistor** above.



PNP **transistor** switch (load is on when IC output is low)



LED lights when the LDR is **dark**



LED lights when the LDR is **bright**

The top circuit diagram shows an LDR (light sensor) connected so that the LED lights when the LDR is in darkness. The variable resistor adjusts the brightness at which the **transistor** switches on and off. Any general purpose low power **transistor** can be used in this circuit.

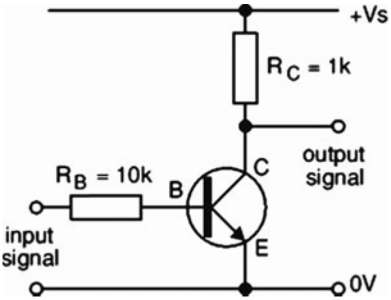
The 10 kΩ fixed resistor protects the **transistor** from excessive base current (which will destroy it) when the variable resistor is reduced to zero. To make this circuit switch at a suitable brightness you may need to experiment with different values for the fixed resistor, but it must not be less than 1 kΩ.

The switching action **can be inverted**, so the LED lights when the LDR is brightly lit, by swapping the LDR and variable resistor. In this case the fixed resistor can be omitted because the LDR resistance cannot be reduced to zero.

A Transistor Inverter (NOT Gate) Circuit

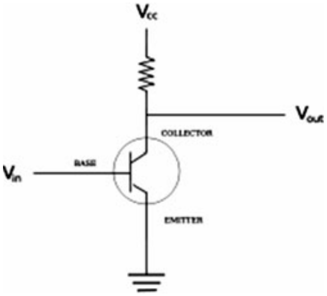
Inverters (NOT gates) are available on logic ICs but if you only require one inverter it is usually better to use the CMOS circuit shown at the end of this section. Note that a single NOT gate is abbreviated by a small circle called a *bubble*. The output signal (voltage) is the inverse of the input signal:

- When the input is high (+Vs) the output is low (0 V).
- When the input is low (0 V) the output is high (+Vs).



Any general purpose low power NPN **transistor** can be used. For general use $R_B = 10k\Omega$ and $R_C = 1k\Omega$, then the inverter output can be connected to a device with an input impedance (resistance) of at least 10 kΩ such as a logic IC or a 555 timer (trigger and reset inputs).

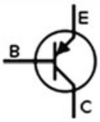
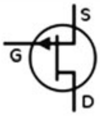
When connecting the inverter to a CMOS logic IC input (very high impedance) one can increase R_B to 100 kΩ and R_C to 10 kΩ, this will reduce the current used by the inverter.

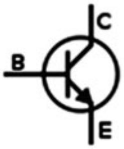
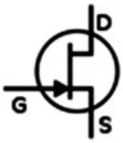


Simple circuit to show the labels of a bipolar transistor.

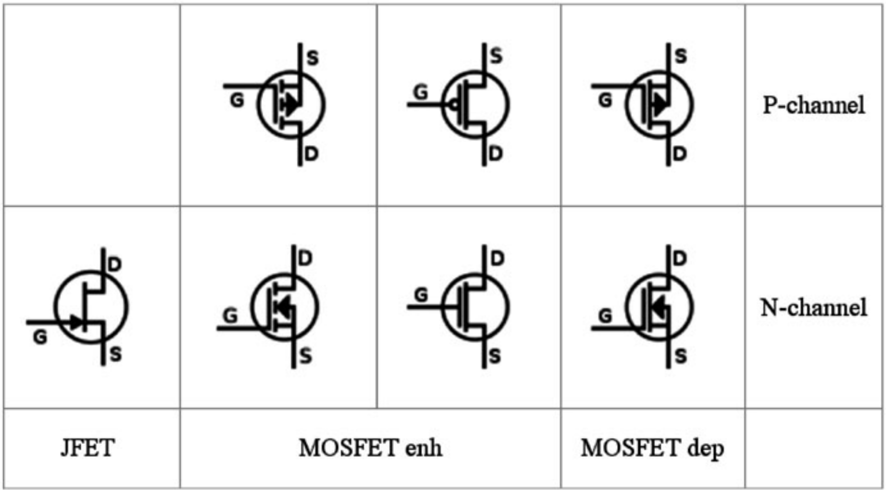
The two main types of transistors have slight differences in how they are used in a circuit. A *bipolar transistor* again has terminals labeled **base**, **collector**, and **emitter**. A small current at the base terminal (that is, flowing from the base to the emitter) can control or switch a much larger current between the collector and emitter terminals. For a *field-effect transistor*, the terminals are labeled **gate**, **source**, and **drain**, and a voltage at the gate can control a current between source and drain.

The images below represents a typical bipolar transistor in a circuit. Charge will flow between emitter and collector terminals depending on the current in the base. Since internally the base and emitter connections behave like a semiconductor diode, a voltage drop develops between base and emitter while the base current exists. The amount of this voltage depends on the material the transistor is made from, and is referred to as V_{BE}.

	PNP		P-channel

	NPN		N-channel
BJT		JFET	

BJT and JFET symbols. Note directions of currents.



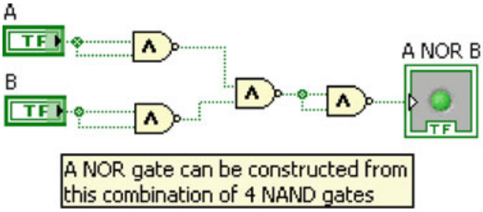
JFET and IGFET symbols
Transistors are categorized by

- Semiconductor material: germanium, silicon, gallium arsenide, silicon carbide, etc.
- Structure: BJT, JFET, IGFET (MOSFET), IGBT, "other types"
- Polarity: NPN, PNP (BJTs); N-channel, P-channel (FETs)
- Maximum power rating: low, medium, high
- Maximum operating frequency: low, medium, high, radio frequency (RF), microwave.

(The maximum effective frequency of a transistor is denoted by the term f_T, an abbreviation for "frequency of transition". The frequency of transition is the frequency at which the transistor yields unity gain).

As proven in Boolean algebra, all types of Boolean logic gates (e.g. AND, OR, NOT, XOR, NOR) can be created from a suitable network of NAND gates and inverters (NOT gates). Rather than draw the symbols for NOT gates a small circle (called a *bubble*) is attached to the output side of other gates as shown in the diagram below. Similarly all gates can be created from a network of NOR gates. Historically, NAND gates were easier to construct from MOS technology and thus NAND gates served as the first choice in Boolean logic in electronic computation.

For an input of 2 variables, there are 16 possible Boolean algebraic functions (For n variables, there are 2↑n inputs. Each input can be assigned 2 possible outputs. Hence there are 2↑(2↑n) different Boolean functions of n variables). These 16 functions are enumerated below, together with their outputs for each combination of inputs variables.

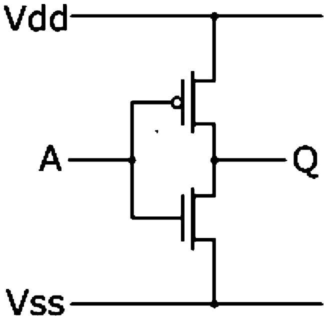


A NOR gate can be constructed from this combination of 4 NAND gates

INPUT	A	0	0	1	1	
	B	0	1	0	1	
OUTPUT	FALSE	0	0	0	0	Whatever *A* and *B*, the output is false. Contradiction
	A AND B	0	0	0	1	Output is true if and only if (iff) both *A* and *B* are true
	A $\nrightarrow$B	0	0	1	0	*A* doesn't imply *B*. True iff *A* but not *B*
	A	0	0	1	1	True whenever *A* is true
	A $\nleftarrow$B	0	1	0	0	*A* is not implied by *B*. True iff not *A* but *B*
	B	0	1	0	1	True whenever *B* is true
	A XOR B	0	1	1	0	True iff *A* is not equal to *B*
	A OR B	0	1	1	1	True iff *A* is true, or *B* is true, or both
	A NOR B	1	0	0	0	True iff neither *A* nor *B*
	A XNOR B	1	0	0	1	True iff *A* is equal to *B*
	NOT B	1	0	1	0	True iff *B* is false
	A $\leftarrow$ B	1	0	1	1	*A* is implied by *B*. False if not *A* but *B*, otherwise true
	NOT A	1	1	0	0	True iff *A* is false
	A $\rightarrow$ B	1	1	0	1	*A* implies *B*. False if *A* but not *B*, otherwise true
	A NAND B	1	1	1	0	*A* and *B* are not both true
	TRUE	1	1	1	1	Whatever *A* and *B*, the output is true. Tautology

The four functions denoted by arrows are the logical implication functions. These functions are generally less common, and are usually not implemented directly as logic gates, but rather built out of gates like AND and OR.

Below is a CMOS circuit for NOT built from two CMOS transistors (with a bubble in one). As explained below, CMOS logic requires more transistors but uses less power and is the logic used on chips.



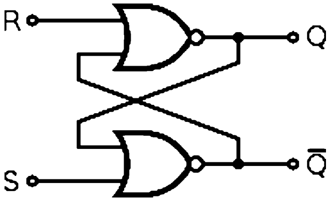
CMOS inverter (NOT logic gate) **Complementary metal-oxide-semiconductor** (**CMOS**) logic is a technique for constructing better-powered integrated circuits. CMOS logic is used in microprocessors, microcontrollers, static RAM, and other digital logic circuits. Frank Wanlass successfully patented CMOS in 1967

In the P-H book, all logic circuits are in integrated circuits and all semiconductors are of the CMOS type. In the Agarwal & Lang book, all semiconductors in gates are initially of the MOSFET type, since it is easy to build gates from them and understand how the gate circuits work. For example, an A NAND B gate is easy to construct with two n-channel MOSFET transistors functioning as switches S and T connected in series to a load, S having input A and T input B. There will be an output of 0 from this series NAND circuit exactly when both A = 1, closing switch S, and B = 1, closing switch T, creating a short circuit from ground to load. To obtain an AND gate from this NAND gate it suffices to place an inverter (bubble) on the output terminal. The new output is then 1 when both A = 1 and B = 1. Similarly, a gate for ANOR B is constructed by connecting two MOSFET transistors acting as switches S and T in parallel to a load. The output is 1 exactly when both A = 0 and B = 0, so that both S and T are open. This is the truth table for NOR (= NOT OR).

The ***load*** *f*or a gate circuit using n-channel MOSFET transistors (NFETs) is usually depicted as a resistor RL connected to the source supply voltage, VS. In practice, load resistors like RL in an IC would take up too much space on the chip. The resistor RL is replaced by a MOSFET with its gate connected to a second supply voltage VA at least one threshold higher than VS. Thus, this load MOSFET remains in the on state for any voltage between 0 and VS applied at its source so that its MOSFET resistance RON replaces RL. This style of building logic gates is called NMOS logic. Unfortunately, NMOS logic gates dissipate static power when the circuit is idle. Therefore, they are replaced by yet another different style of gate logic called CMOS (Complementary MOS) which has very low static power dissipation. CMOS logic gates require an extra *complementary* p-channel MOSFET (called a PFET). Which acts in a manner complementary to the basic NFET in the gate. When the gate-source voltage VGS of the NFET is greater than a threshold voltage, the NFET turns on and a resistance RONn appears between its drain and source. In contrast, the PFET turns on when its VGS is less than a threshold and then a resistance RONp appears between its drain and source. Provided that the gate input voltage vin is at VS or 0, the NFET and PFET transistors are never on at the same time under static behavior so that there is never a resistive path from the power supply to ground. Hence, there is no static power dissipation. As an example, the earlier diagram above is a CMOS logic gate for NOT (i.e. an inverter).

A common logic circuit is the RS (Reset-Set) *flip flop* (or *latch*) shown below. It has two stable states and can therefore store a memory bit in either the 1 or 0 state depending on the R and S inputs.

RS (Reset-Set) Flip-Flop



An RS latch, constructed from a pair of cross-coupled NOR gates

RS (Reset-Set) flip-flop

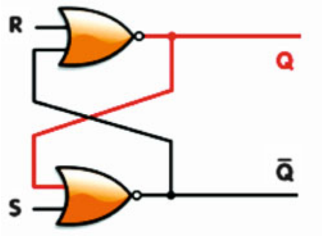


Illustration of RS latch operation. Red and black mean logical '1' and '0', respectively.

The fundamental latch is the simple RS flip-flop (also commonly known as SR flip-flop), where R and S stand for reset and set, respectively. It can be constructed from a pair of cross-coupled NAND or NOR logic gates. as shown here. The stored bit is present on the output marked Q.

Normally, in storage mode, the R and S inputs are both low, and feedback maintains the Q and $\bar{Q}$ outputs in a constant state, with $\bar{Q}$ the complement of Q (Simply check the NOR truth table when $Q = 1$). To store a bit on a new clock cycle, if S is pulsed high (set) while R is held low, then the Q output is forced high, and stays high by feedback even after S returns low; similarly, if R is pulsed high (reset) while S is held low, then the Q output is forced low, and stays low even after R returns low.

The next-state equation of the RS flip-flop is

$$Q_{next} = S + \bar{R}Q$$

where Q is the current state. Q_{next} becomes Q (the new stored value) at clock edge.

*This equation originates from C. Shannon's 1937 master's thesis, A Symbolic Analysis of Relay and Switching*Circuits

Appendix 2

Hardware for the User Interface

The modern computer devices that most readers are familiar with are a personal desktop computer (pc) or workstation, a laptop or a high-tech cell phone. These devices, as usually assembled, have two main components:

1. a box, or compartment, in which are housed the hardware electronics (chips/circuit boards, memory modules, individual transistors etc. as described in the preceding Appendix) which execute the computation operations and control the sequence of steps in a computation and
2. a second component consisting of the physical hardware parts which interact with the user, usually a *monitor*, *keyboard* and *mouse*.

The Monitor as a Visual Output Display

Most of us are familiar with the monitor connected to a pc. Originally, it was a cathode ray tube of the kind commonly used in laboratories in oscilloscopes to provide images of x-y plots on a coated fluorescent screen illuminated by a moving electron beam. The screen was calibrated as a rectangular grid of points. The modern monitor is a similar device but uses a liquid crystal display (lcd) screen as in many television screens. The lcd screen is fabricated as a discrete rectangular array, for example 1,920 × 1,280, of individual liquid crystal picture elements called *pixels*. A pixel crystal will not transmit a beam of light or electric charge unless it is properly oriented. Its orientation relative to the beam is controlled by an applied current passing through a tiny transistor at each pixel. Thus a monitor has an associated data array of pixel currents. The values in this array of applied currents may consist of one or more bits to control the brightness of a pixel, making it visible to the user. Color is achieved by having three crystal elements (red, blue, and green) at each pixel. This array of pixel data is called a *bit map* and is controlled by a software program that provides a bit map matrix for the pixel data array and other screen properties and is known as a graphical user interface (GUI). By careful detailed programming of the bit map, a GUI can create moving complex images on the monitor display, as output from a running application program.

To allow user interaction with the monitor display an engineer named Engelbart invented the device we know as the *mouse*. The mouse controls a *cursor* (usually an arrow symbol) on the screen. The cursor has a position (monitor coordinates) that changes as the mouse is moved on a pad vertically and horizontally by the user. The mouse also has one or more switches which are actuated by the user pressing buttons on the mouse, such switch actions being called *clicks*. A *click* or sometimes a *double click* sends a signal to the GUI which elicits a response from the GUI, such response being programmed to give the user a specified action on the monitor screen depending on the local part of the screen image pointed to by the mouse cursor. For example, local parts may be icons representing various files or other computer elements to be processed or they may be labels of a menu of alternative actions which the user can select by pressing the mouse button. These conceptually simple user-computer interactions provided by the active monitor display as output and the mouse position and clicks as input have made possible a rich environment of computer usage. More sophisticated user interactions are currently provided by tactile hand motions on capacitor-sensitive monitor surfaces as, for example, in "smart" phones like the Apple iphone and the RIM blackberry phone.

Chapter 6
Operating Systems (OS)

Edward K. Blum

In Chap. 4 (Software), the *user interface* (interaction) with the computer hardware (Chap. 5) is implicitly specified as being via a compiler program which translates user-defined high-level language programs (e.g. C++ programs) into low-level assembly language programs. The latter are parsed into a sequence of simple machine-like operation-based statements (like X = Y + Z) that can be executed more-or-less directly by MIPS-type hardware computers configured with registers and processors, as explained in Chap. 5. This application-software (e.g. C++) interaction with the computer hardware, the *user interface*, is supported by little more than a hardware device called a *program cou*nter in conjunction with straightforward state-control logical circuits, following Turing's basic ideas on computation Control rather closely. However, in this early method of treatment of the rather simple user interface, there is no account given of certain implicit and important details, such as how the C++ user gains access to the compiler or to other assumed supporting facilities such as library subroutines called in the high-level user program or to data-file storage. For the early instances of software-hardware interactions, this naive approach to the user interface was sufficient. However, by the 1960s, hardware and application software programs had both become much more sophisticated and the user interface more complicated.

As observed in Chap. 5, the invention of the transistor and integrated circuits produced great increases in execution speeds, clock rates in logical circuits jumping from megacycles (10^6 cycles)/s to gigacycles (10^9) cycles/s. Furthermore, data storage devices such as magnetic discs had likewise become much improved in capacity as well as speed of data access. Many user application programs began to manipulate large files of data as one of their main activities (See Databases Chap. 10). High-level languages (e.g. C++) provide data management operations for user

E.K. Blum (✉)
Department of Mathematics, University of Southern California, Los Angeles, CA, USA
e-mail: blum@usc.edu

E.K. Blum and A.V. Aho (eds.), *Computer Science: The Hardware, Software and Heart of It*, DOI 10.1007/978-1-4614-1168-0_6,

programs, such as file-manipulating operations (e.g. opening and closing *sequential* files) which must be supported by a *file system* (A *sequential* file is a set of data elements, such as bytes or words, arranged in a sequence so that a data element at position n in the sequence can only be read/written by first reading/writing the preceding n-1 elements, as on a magnetic tape or disc. By contrast, in a *random access* file a data element can be accessed directly in a fixed time interval without going through the other data elements, as in an internal electronic memory device).

A *file system*, as the word *system* suggests, embraces both software and hardware. It includes software specification of file data formats (beyond mere bits and bytes) and software specifications of *file structures,* say as individual *records* with their own identifiers, these software specifications being supported by associated hardware for storage and accessing of large data files on magnetic discs (*hard drives*) (See Chap. 8, Databases). As computer usage grew and encompassed multi-user large central computers, computation became a more complex and multi-faceted activity with technological and operational problems having to do with modes of computer usage rather than with the internal computations. Practical problems arose in running application programs in situations not anticipated by the early pioneering mathematicians and engineers. As the applications of computers expanded in number and variety, the early simple user interfaces evolved into a more complex phenomenon as important as the basic application computations and placed practical demands on computer systems for additional supporting facilities for the interfaces.

As noted, the use of the word *system i*n the compound term *file system* entails a combination of software and hardware support. The practical value of various user interface support facilities providing services beyond those for explicit hardware-software interactions was soon recognized. This led to the realization that many other interface support provisions were needed for large fast computing applications, especially on central shared computers, and the support should be integrated and organized into a broad system entity known as an *operating system* (OS).

An OS is certainly needed for multi-user access to a central computer. Even on single-user personal computers the user requires such services as control of printer output, word-processing and management of files. The OS approach to computation on electronic computers was soon widely accepted and implemented by manufacturers like IBM and Honeywell. In fact, each computer is now organized to be used through the intervention of an *operating system.* The OS is supplied in the form of a *systems program* installed in the computer memory together with special hardware devices so as to act as the interface between a user and the computer. Three classes of OS systems currently are dominant:

1. various Unix (or Linux) systems on mainframe central computers;
2. Windows systems on most personal computers (pc's);
3. MacOSX systems on Apple computers.

The Unix OS was initiated at ATT Bell Labs by Ken Thompson and developed further, along with Dennis Ritchie, using the C language designed for that purpose. They received a Turing award for their efforts. A popular version of Unix, called BSD Unix, was developed at the Berkeley campus of the University of California.

The OS called Windows is a product of Microsoft Inc. and is used on most pc's other than Apple Mac's which use versions of the OS called MacOSX, which is related to Unix in many features.

As remarked above, along with the development of large and fast data storage devices as part of the hardware came the steady and rapid increase in speed of the CPU and other electronic components in computer hardware. As a result, running a single user application program on a computer often consigned the fast CPU to an idle state as the computer arranged data for execution of the next program statement. Likewise, increased speeds of data input and output (IO) in storage devices gave rise to idle states in these devices as they waited for lengthy computations to be completed. To improve the efficiency of utilization of a large computer a mode of multi-user access to a single computer was adopted in which many user "jobs", as they are called, were allowed to run partly "concurrently" in a *time-sharing* (or *multi-tasking*) mode whereby time is divided into *time-slots* and computer resources are allotted to different jobs in each time-slot. As one of its support functions, the operating system *schedules* the allocation of time-slots to each user, say on the CPU or on file storage devices. The *scheduler* program in the OS tries to optimize utilization of computer hardware resources by minimizing idle time while not appreciably slowing down the execution of the sharing individual user jobs. This scheduling problem arises in non-computer industrial "job shops" where resources are time-shared and has been treated by operations research methods. The OS scheduler program can avail itself of these methods, but the scheduling problem may not be amenable to practical exact solutions under some conditions on the flow of jobs submitted.

These additional user interface supports, and others, eventually made the OS software a complicated *system program*. We shall illustrate this complication by summarizing below how the OS program manages the file system, one of its important functions. Before doing so, we must consider a fundamental question for an OS, namely, just what is its user interface? Since the OS often manages user access to a multi-user central computer, an explicit user interface must be provided.

The OS Kernel and the Shell

Since the OS program essentially supervises how user *jobs* (application programs) are executed, it must interact directly with the hardware, for example in a MIPS computer by supervising compiled assembly language statement execution by moving data items into and out of registers and the CPU, as explained in Chap. 5. To run smoothly and avoid errors this segment of the OS program must be shielded from direct user interference. Likewise in a multi-user central facility certain parts of the OS must be shielded/protected from user interaction. For a Unix style OS, to accomplish this the OS program is organized so that an "insulated" or "shielded" mode of execution of the hardware-interactive segment of the OS program takes place in what is called its *kernel*. The *kernel program* interacts directly with the hardware (say manipulating internal registers) and contact with users is avoided by various protective devices, possibly by having the kernel program stored in *hardware-protected* memory locations which are not accessible to users.

However, it is necessary for a user to communicate with the OS to request various *services*, such as compiling a user program or providing file system actions. In effect, a user is a *client* and the OS is a *server* which responds to client messages requesting services. In Unix systems, these messages are in the form of *shell commands* issued by the user in a user-accessible part of the OS known as the *shell*. As the name suggests, the shell is a program or *process* which acts as the interface medium between users and the OS kernel. The *shell command*s allow users to contact the kernel to request and receive OS services while maintaining the shielded execution mode of the kernel.

The shell is made accessible to each user as part of the user initial login process. As stated above, for modern computers a user must use the OS to access the computer. The user begins by obtaining a user *account* with a *user's name* from the *administrator*. A user then *logs in to* the user's account, possibly at a remote console or workstation connected to the computer OS. If *login* is accepted by the OS, the OS login *process* sends a *shell prompt* symbol back to the user console signaling that the shell is ready to receive commands from the user as a client and respond to them as a server. In Unix (and Linux) the shell commands have the format.

Command_name opt1 arg1 [opt2, arg2, . . .,]

where the opt's are options which modify the command behavior and the arg's are arguments which provide data needed by each option.

In Unix, there are two commonly used shell processes: (1) the Bourne shell from Bell Labs and (2) the C shell from Berkeley, designated as *csh*. In Linux, the default shell is *bash*, designating the variation called the *Bourne-again shell*.

A shell is a part of the OS system program which receives and interprets user commands and then interacts with the kernel as needed. It is able to penetrate the protective shields of the kernel program. To further explain this rather sophisticated OS behavior, we shall give examples of shell commands which deal with management of the file system, which is a major part of the OS.

File Systems

In Unix and Linux, the *file system* is organized in a logical structure that can be depicted as a rooted tree, TR say, which is mathematically a *directed graph* (i.e. a graph with edges that have directions) shaped like a rooted tree wherein the nodes can be files of data, or *directories* of files or *subdirectories*. A *directory* is an index (or pointer) which locates a root of a subtree of TR. The index becomes part of a *file path* name which can be used by the OS to navigate through the nodes of the tree TR in a natural way to reach a designated file node. Directory nodes usually have many outward edges leading to multiple file nodes. In the Windows OS, the directories are called *folders*, suggesting collections of related files. A file node usually has one outward edge which determines the location and size of a (sequential) file of data in the hardware.

In Unix, the root of the entire tree TR is a directory denoted by the slash /. This is the start point of an overall search of TR for other nodes. Each user account is assigned a *home* directory which has the same name as the user account and is designated as /home/user_name. All files and directories created by the user are stored in the user home directory unless otherwise explicitly indicated. To access them the OS uses a *filepath_name* starting with /home/user_name/. As an example, consider using the shell to create an empty file named empty.text. This can be done by typing the shell **touch** command at the shell *prompt* symbol ([. . .]$) as follows:

```
[console1 user_name]$ touch empty.text
```

To list the file, type the shell **ls** (list) command as follows:

```
[console1 user_name]$ ls
```

The shell responds with

```
[console1 user_name]$ empty.text
```

Since no directory names were given as arguments in these commands, the shell assumes that they refer to the user's *current working directory (**wd**)*. The shell takes **wd** as the default when no directory name is given in a command. **wd** is assumed to be the home directory unless explicitly changed by the **cd** command. For example,

```
[console1 user_name]$ cd /
```

changes the working directory to be the root directory /. The new prompt will be

```
[console1 /]$
```

To see that this has taken effect we can use the **pwd** (present working directory) command to get the following display:

```
[console1 /]$ pwd
[console1 /]$
```

To return to the user home directory simply issue the **cd** command without arguments. Some other file system commands are as follows.

To copy a file such as empty.text into a new file named backup.text use the shell **cp** command as follows:

```
[console1 user_name]$ cp empty.text backup.text
```

To move a file from one directory to another use the shell **mv** command. To delete a file use the shell **rm** command. To delete a directory use the shell **rmdir** command. These and other Unix file-system commands are explained in the book "Teach Yourself Red Hat Linux 8" by Aron Hsiao published by SAMS, Indianapolis.

They illustrate how a user interfaces with the Unix OS by issuing shell commands at a console. The shell interprets the commands and translates them into requests for services by the kernel and passes them to the kernel. For example, the kernel will create a file on a hard drive when requested by a user **touch** command and keep track of its location so that it can delete it when requested by a user shell **rm** command. Of course, file IO operations can also be specified in the application program itself and be done during execution of the job; e.g. in C or C++ there are file operations to **open** and **close** named files, which become nodes in the tree TR. When a declared named file is opened, the application program can then write into it using the **fprintf** operation. Likewise it can read a file that is opened by using the **fscanf** operation. For computations which produce a large output data file by iterations using **fprintf**, say into an array, the user may not wish to read the entire array file during execution since this would slow down execution. After the job ends, the user may then wish to examine the output data file produced by the OS. The user can do this using shell ls commands to locate the file, say filename, as described above. The shell command **cat** filename will display the contents of filename at the console. There are also commands to request a printer service. Thus, the command

[console1 user_name]$ **lpr** filename

creates a print job in the printer queue which will cause printing of the contents of filename when the job reaches the front of the queue.

Note that the shell also provides a command to compile a source program file Prog created by using an *editor* program such as Emacs. Prog is translated into an *object code* file, say prog.obj, by the compiler. To run prog.obj it may also be necessary to *link* prog.obj to other object code programs provided by the OS. After compiling and linking is completed the shell places prog.obj in a directory /bin containing *executable* files. A shell command which simply references prog.obj then requests its execution by the kernel.

To allow its time-shared execution an executable file like prog.obj is restructured as one or more *processes* (We have already referred to various OS processes above). To convert an executable file into a process it is necessary to determine which memory locations are needed as a *local state* of the computation. The data in these locations must be saved in a block of memory associated with prog.obj so that its execution can be resumed after being paused when its allocated time-slot expires. See scheduling below.

File Permission

A file has one or more *owners,* various users who can specify *permissions* to apply file operations to the file. The files in the home directory are owned by the user and also possibly by a user group created by the system administrator. The group includes the user who created the file. An entry in the file node specifies a list of permissible operations the user group can perform on the file. A typical entry in /home might be d rwx xr r x. The letter d denotes a directory. w means permission to write. r is permission to read. x means permission to execute. The command **chmod** can change permissions.

Using the shell is a powerful way to make requests for OS services. To facilitate it the OS provides for shell *scripts*, which are small programs consisting of shell commands as basic statements. A script is created as a file by a text editor such as emacs. On the first line the user types #1/bin/sh to indicate that a script follows. Scripts can be written as named subroutines. Script statements can involve variables in assignment statements. The values of variables are supplied in script calls as in subroutine calls. A script can use an if...then...else statement to alter control of execution of commands. The while statement is also allowed.

Useful commands permit redirecting output of other commands. The ">" character redirects standard output to the monitor to a file. Thus, **ls > dir.list** redirects output normally sent to the monitor to the file dir.list. To append rather than overwrite output use ">>". Output can also be directed to be the input to another command by using the **pipe** character "|" as in **ls|more** which permits display of a long list of files on one monitor screen.

Scheduling

We have already referred to the scheduling of jobs performed by the O*S*. In fact, an OS program has many jobs of its own which perform services to users; e.g. the job that implements the shell lpr command for printer activation. The scheduler must give this job repeated execution time slots to provide timely responses to user requests for printing files.

To facilitate the OS scheduling function it organizes jobs into (usually small) segments called *processes*. A *process* consists of a file of a section of executable statements of the job's compiled program and blocks of data encoding the local state of this section of the job's computation. For example a section can consist of a short sequence of a few compiled statements to be executed in a time-slot. The variables originally used in these statements have been replaced by memory or register locations. These locations constitute the "local state" of the computation. They contain the information needed to resume the computation after the job is temporarily paused by expiration of its time slot. Processes are queued for execution by the OS scheduler and their saved local states are restored in an "active" data block for execution by the kernel when the next time-slot is allocated to the job. Details of the scheduler queueing discipline vary from one OS to another. For example, they may involve job priorities assigned by the system administrator (system jobs like lpr receiving high priority), lengths of job execution times as estimated by the users and other job properties affecting overall access to the computer resources. These details affect system "overhead", the costs entailed by the *swapping* of job processes into and out of the execution queue.

Concluding Observations About OS's

Operating systems are usually large and complex *systems-type* programs which manipulate computer registers, files and other data structures. As stated above, the Unix OS was written in the C language, which has useful machine-oriented operations.

The preceding pages summarize the main role of an OS in managing multi-user access to a central computer. The reader can obtain further details by consulting the user manual usually provided by each computer facility. For the various Windows systems for pc's. The reader can consult user manuals published by Microsoft Inc. or use the built-in HELP command on the pc to print detailed OS information on the pc monitor screen. For Unix/Linux OS's there are many published books. A few references are listed below.

References

SAMS Teach Yourself Red Hat Linux 8 , Aron Hsiao 2008 Sams Publishing, Indianapolis, Indiana.

The Unix and X Command Compendium A. Southerton and E. Perkins 1994, John Wiley, New York.

Linux Configuration and Installation P. Volkerding K. Reichard and Eric Johnson 1995 MIS Press Henry Holt Inc. New York

Unix System Administration Handbook E. Nemeth G. Snyder and S. Seebass 1989 Prentice Hall Englewood Cliffs, New Jersey

Chapter 7
Computer Networks

Fan Chung Graham and Edward K. Blum

This chapter on Computer Networks covers one of the most influential developments involving Computer Science in the past two decades, the amalgamation of two major technological fields: Computation and Communication. As was to be expected from its title, this book discusses various aspects of Computation in chapters dealing with the processing of information at a particular locale by a single computer. Perhaps surprisingly, this chapter on Computer Networks considers the transmission of information across the globe and the processing of this information, perhaps in a multifaceted manner, at computers distributed over widespread locations. Another Chap. 9, describes a different variety of distributed computing in a different context. Still another Chap. 8, describes the distribution of computing over a cluster of thousands of pc's situated in a local network at essentially one location. It also briefly describes cloud computing. Distributed computing has become a major phenomenon in computer science.

Perhaps the content of this chapter is not really so surprising to those readers who are aware of the widespread and burgeoning use of email and text messages sent by computers over the *Internet* and *World Wide Web* and the proliferation of "smart" cell phones that are really small computers that communicate. This kind of computer activity, called *computer networking,* has become so common that the word "google", referring to searching for information on *web pages* by means of the computer system known as the Google "search engine", has become a verb in everyday language. How did this per-vasive phenomenon of computer networking arise? How did the two scientific disciplines of Computation and Communication amalgamate to yield a new discipline, *computer networking,* which extends beyond

F.C. Graham (✉)
University of California at San Diego, La Jolla, CA, USA
e-mail: fan@ucsd.edu

E.K. Blum
Department of Mathematics, University of Southern California, Los Angeles, CA, USA
e-mail: blum@usc.edu

E.K. Blum and A.V. Aho (eds.), *Computer Science: The Hardware, Software and Heart of It*, DOI 10.1007/978-1-4614-1168-0_7,

scientific domains and penetrates social aspects of life? In what follows, we shall address these questions to give the reader some background understanding of this Internet phenomenon and then we shall give an overview of how the Internet works, especially for computer networking.

In Chaps. 2 and 3, some of the history of Computation is presented. It is shown that the modern aspects of Computation and Computer Science began in the 1940s with research on software and the building of early versions of computer hardware. By contrast, Communication technology is an older field, fairly well-established by the 1940s. If we limit our attention to electromagnetic (EM) communication, we can reasonably assert that this field started with the transmission of electrical signals along copper cables in the 1850s. For the transatlantic cables of that era, this process is governed by the partial differential equation known as the Kelvin Cable Equation (e.g. see the book Mathematics of Physics and Engineering by Blum and Lototsky (2006)). For the somewhat earlier cases of EM communication, the transmission of telephone signals (Alexander Graham Bell) and telegraph signals (Samuel F.B. Morse) across land areas, ordinary copper wires (strung on telephone poles) allowed propagation of EM signals according to the telegraph equation (Again, for engineering/scientific details see Blum and Lototsky, pages 333–335.) Eventually, communication engineers (e.g. RCA radio engineers) learned to transmit signals without wires by electromagnetic (EM) waves traveling in space, by means of a *wireless* system of radio transmitters, receivers and antennas. The existence of EM waves in space was unknown until remarkably postulated by Clerk Maxwell and actually implied theoretically by Maxwell's equations in 1864. (This is one of the profoundest examples of the predictive power of mathematical models of physical phenomena. See the Blum and Lototsky book for technical details.) To establish that Maxwell's prediction and theory of EM waves in space is physically realizable the physicist Hertz actually generated such waves in a laboratory experiment some 20 years later. It was further postulated by Maxwell and verified by experiments that EM waves can propagate in a vacuum at the speed of light, which itself consists of EM waves of various frequencies. Nowadays, computer networks make widespread use of wireless EM wave connections both for *digital* data transmission (of text) and *analog* signal transmission (of voice and video signals).

In the early days of EM communication, networks of wires were built by companies like ATT (American Telephone and Telegraph) to connect many telephone users to each other. The technology of these *telecommunications* networks was based on a system of central *Exchanges* to which user wires were connected (along telephone poles), and in which were installed large switch mechanisms. Individual users were connected to an Exchange switch mechanism and a system of user numbers allowed the switches to connect one user caller to another user respondent. The Exchanges also amplified signals which needed reinforcement after decay due to transmission losses. Initially, switches were human-operated (by "operators" manually connecting sockets in an Exchange by plugging in wires). Later, switches were fabricated as electromechanical relays which were actuated automatically by circuits acting on the incoming signals. Still later, relays were replaced by transistor switches. (See the Appendix to Chap. 5 on logical circuitry.)

The Internet

As the population of phone users grew in the first half of the twentieth century, single Exchanges could not practically handle the volume of individual telephones despite large assemblies of sophisticated switches. It became necessary to establish *networks* of Exchanges at different geographical locations and to connect the Exchanges by building higher-levels of super-networks. This was done simply at first by dividing a city or other region into geographical areas each with their own numerical *area code*, which codes became appended to the user numbers, as they are today. Signals were *routed* from individual phones to a local area Exchange in a *core* network and thence through a "long distance" network of connected area Exchanges to a destination Exchange having an area code keyed in by the caller and thence to the called phone number.

As telephone usage continued to grow, a topology of hierarchies of networks of networks and routing protocols connecting various levels of networks evolved and was carried over to the computer networks that came later and exist today. Companies besides ATT (e.g. Verizon, Sprint, Time-Warner) now provide thousands of *core* networks, inter-connected by "long-distance" networks often employing fiber optic cables carrying light waves to achieve greater *bandwidth* (bits per second transmitted), and to which "local carrier" networks can connect. Further, wireless connection networks are now providing transmission of electromagnetic wave signals through space between radio towers for cell phone communication. The use of wireless networks proliferated as more devices activated EM radio wave signals. Meanwhile, the network transmission of text data messages between computers began and quickly increased in volume and was supplemented by the transmission of accompanying voice and video images. Many manufacturing companies began producing hardware for network communication, introducing their own engineering methods. A variety of methods (*protocols*) for routing messages through a multitude of networks was implemented. The result is a complex heterogeneous *global network* transmitting *digital* data messages and *analog* signals. Gradually, under pressure of government agencies and industry self-regulations, there evolved a standardized set of basic *protocols* (formats of data and transmission procedures) for transmitting *analog* (voice and video) signals and *digital* text data in core networks and between these networks at distances involving hierarchies of what can be viewed as large switch Exchanges as described above. Communication paths were extended by means of a hierarchy of network interconnections to form the vast global network that is now known as the *Internet.*

The Internet

The I*nternet* is a global system of wire transmission lines and wireless EM transmission towers that connects a computer or other device (such as a phone) first directly to an individual core network (e.g. an ATT or Verizon or Sprint network of phone wires and wireless links or a Time-Warner cable network) and then through a hierarchy of inter-network connections to destinations throughout the world today. The companies that own and maintain the various communication core networks and parts of their inter-network connections are known as Internet Service Providers (ISP). Users gain access to the Internet by subscribing to a service by an ISP through a phone line, cable line or wireless (antenna) connection, and from the ISP's network further access is provided to the global Internet through network-level interconnections to networks located throughout the world.

There are now thousands of core networks distributed over the world and manifold means (*protocols*) of connecting them to each other and to individual users. As in the early telephone networks, connections are done by means of a system of network *addresses* analogous to the early telephone numbers. However, the address system of the Internet is necessarily more complicated. As we shall see, addresses consist of two parts: (1) user identifiers (such as names of *host* computers or the NIC numbers in an ethernet host described below) and (2) network identifiers analogous to the area codes in telephone networks. Although there is no single geometry of network connections many small networks are either in a star configuration (a central node with branches to all other nodes) or in a ring shape (a closed path of nodes).

Within the physical Internet, a system of "websites", with installed software called "web pages", located at various points in the Internet sprang up to offer global information services in what is known as the "world wide web" (WWW or simply "the web"). The web is the "brainchild" of a physicist, Tim Berners-Lee, who was one of the first to see the information-disseminating/gathering potential of websites. Besides information exchange, the impact of the web on our modern lifestyle has been incalculable, notably in virtual *social* networks (e.g. Facebook and Twitter) which have captured a vast clientele of active users.

Graphs of Networks

It is convenient to represent the geometric connectivity of a network by a mathematical structure called a *graph.* A graph can be depicted as a set of points, called *nodes,* (in computer-communication networks representing communication or computer devices like phones, computers, Exchanges etc.) and a set of lines, called *edges*, shown as joining certain pairs of nodes, each edge representing a physical link (wire or wireless connection) joining the pair of nodes in the network. A *path* in a graph joining two nodes A and B is a sequence of adjacent edges joining A to B in the network. The combinatoric properties of graph paths can be quite complex, reflecting the geometric complexity of communication paths in the network. There can be many paths joining two device nodes A and B, say by different routes through Exchange nodes. Algorithms for finding optimum paths (e.g. shortest or least busy) are among the topics treated in the graph theory of networks. . A brief survey of graph theory is given in an Appendix to this chapter. It amplifies the following brief introduction.

Graph Theory

Graph theory and computing are intertwined in numerous ways. Various special graphs, such as spanning trees (tree shaped structures which touch all nodes) and Hamiltonian cycles (closed paths which pass through all nodes once), and various graph properties with natural names (*max-cuts, flows, min-cuts, graph coloring, graph packing* etc.) are main topics in the study of computation and data structures. Facing the challenge of dealing with problems arising in computer networks of tremendous sizes and complexity, many areas in graph theory have been stimulated, enriched and advanced. In particular, combinatorial probabilistic methods and *spectral* methods have been playing an increasingly important role.

Efficient randomized algorithms rely on the use of randomness in graphs, that is, on such probabilistic concepts as *random walks* along paths wherein successive edges of a path are chosen probabilistically at nodes where there are several possible choices of the next edge. Instead of focusing on random graphs with the same probability distribution on each node and its edges as in classical random graph theory, a general random graph theory has been developed for graphs with any given *degree* distribution (of the number of edges at nodes). Instead of previous focusing on a *diffusion* flow to generate paths on lattices or structured graphs, there is now a need to consider *percolation flows* in any given host graph, such as the contact graphs in the study of spreading diseases. In addition to random graph models, it is of interest to quantitatively analyze properties that a random graph satisfies with high probability. There has been extensive usage of *expander* graphs which can be mainly controlled by eigenvalues (Hoory et al. 2006). Of particular interest is the study of random walks (or Markov chains) on graphs. Random walks are closely related to statistical sampling and sampling can be used for designing approximation algorithms (Jerrum and Sinclair 1989). Thus random walks are a useful tool for designing robust and efficient algorithms for searching for desired nodes as seen in many network applications such as specified Web search, social networking, graph sparsification and network games (Nisan et al. 2007). Many new research directions remain to be explored. The Appendix covers some of these. Some immediate references for graph theory follow: Chung (2010), Hoory et al. (2006), Jerrum and Sinclair (1989), Nisan et al. (2007)

The World Wide Web (The Web)

The web is, in a way, a software counterpart to the Internet hardware. It is a collection of nodes, called *websites*, in the Internet containing associated data called *web pages* which can be accessed by a computer program called a *web browser*. The web pages are regarded as *resources* in a vast information library and each page is assigned a unique library address called a Uniform Resource Locator (URL) that can be used by a browser to access the page. The readers of this book may have used the well-known browser for pc's, Microsoft's Internet Explorer. Another well-known browser, for

Mac computers, is Apple's Safari, available also on the iphone. Browsers employ special programming devices provided in a special language called *HyperText Markup Language* (HTML) together with associated computer implementations known as Hypertext Transfer Protocols (HTTP). (See Savitch 2001, Chap. 13.) HTML is a declarative type of programming language, as opposed to the usual procedural languages like C++ and Fortran. The main HTML programming mechanism for performing browsing of Internet webpages is the *hyperlink* statement. This is an *active* statement in the sense that it can be displayed, usually partly underlined, on a *source* computer monitor screen and then activated by a mouse click on the underlined statement text. The click causes a *link* connection to be executed, using HTTP, from the source computer to the webpage specified by the URL in the statement text. These *links* make requests for services from the webpage which are processed in a *client-server* mode, the webpage as server and the source as client. (See below.)

Initially, browsing involved only text messages, but very soon graphic (pictorial) information was involved, as for example in the transmission of video signals. This required greater communication channel bandwidth as the connections between network nodes were required to provide a service called *video streaming*, which is real-time transmission of information (i.e. with no delays for storing data).

Protocols: The Technology of Network Transmission of Messages

In the initial stages of the Internet's evolution, as explained earlier, the telecommunication (voice and then video) *protocols* technology using hardware devices to route messages through networks was developed by phone and network companies and was dominant. But software protocols began to be used for transmission of digital data between computers. With the rapid increase in volume of computer data transmission (e.g. email, file transfers, www data), the protocols for these two types of communication technology converged into a common standard protocol that was primarily implemented in software but with some hardware assists. A protocol called *Internet Protocol* (IP) became the common Protocol of choice for both types of communication. IP was combined with a Transmission Control Protocol (TCP), to form a suite of standards called TCP/IP which was widely adopted as a *stack* of major protocols in response to recommendations of the Internet Society, a professional society of Internet experts, working through its Internet Engineering Task Force. We shall give a summary overview of TCP/IP software protocols below from the perspective of computer data communication. We shall only mention briefly the use of these protocols for telecommunications, as we regard this as more a part of communication engineering rather than computer science. Nevertheless, we shall keep in mind a broader view of the Internet as not one large homogeneous data transmission network but rather as a system of thousands of networks of various types, such as the core networks mentioned earlier and LANs of computers, all interconnected in hierarchies of networks of networks. The initial organizational principle of telephone networks with area Exchange nodes connected in super-networks has been carried over to the Internet in a modified and extended form in which the old Exchanges are replaced by hardware devices called *routers*. A router is located at a network node and has an address which allows it to receive messages from other routers or from computer host nodes in the Internet. A router is installed by a local network administrator with tables of addresses of other nodes in the network to which messages can be forwarded to reach an ultimate destination address.

In about 1977, to further standardize network protocols the **International Standards Organization** (ISO) proposed an overall seven-layer model of network message processing called the **Open Systems Interconnection** (OSI) model which serves as a rough but not mandatory guide to the implementation of computer networking, including TCP/IP. Before we present the OSI model it is useful to retrace some of the early development steps in computer networking.

Implementation of Computer Networks – The Ethernet

Let us go back in time and retrace the development steps in the computer networking phenomenon and ask how it came to pass. How did computers, (the desktops, laptops and hand-held "smart phones") come to supplement and even replace ordinary telephones as a user device of communication? Computer engineers early recognized that computers can be connected in local area networks (LAN's) to collaborate on large computations in a distributed manner or simply to transmit data files to each other. The first widespread computer communication network technology was the *ethernet*, a system which interconnects a LAN of computers by high-quality cables. The ethernet was developed at several places in the 1970s but one of the principal developments was at the Xerox Palo Alto Research Center (PARC), where it was named "ethernet" in a report authored by Dr. Robert Metcalfe. An alliance with Digital Equipment Corporation and Intel Corporation then provided the hardware for standard 10 megabit/s (10 Mbs) local ethernet networks in which data was transmitted at the 10 Mbs rate.

In an ethernet network, communication of digital data, say from a pc in a LAN to other pc's is done by a two-way (send–receive) connection of the pc to a cable. The connection is made through terminals on a circuit card called a *NIC* (*network interconnection card*) installed in a slot on the computer *motherboard,* the terminals being connected to a high-quality ethernet cable. Each NIC is given a unique address when it is manufactured. The NIC is under control of commands issued by the pc central processor. The data in computer send/receive signals processed by the NIC are coded as sequences of voltage pulses modulating a carrier signal and represent digital messages originally coded as sequences of binary *bits* ($0'$s or $1'$s). Since several computer *nodes* in a LAN are usually connected to a single ethernet cable, there can be a resource-sharing contention problem for the cable. The nodes compete for connection to the cable. The NIC handles this contention by what is known as a CSCD protocol in which the Carrier is first Sensed (CS) to determine if another NIC has begun sending a signal on the cable, in which case the NIC seeking to use the cable waits for a specified time delay. If two NICs find the cable to be free they are able to begin sending at about the same instant, but then there is a possibility of collisions of traveling pulses from the two NICs. The NICs detect collisions (CD) by measuring higher voltage levels of the pulses, in which case one of the NICs ceases its transmission. A queueing protocol allows *fair* access of contending computers to the cable, that is, every computer node gets a turn to connect.

This is one way computer-communication is handled at the *hardware layer*.

The ISO Open Systems Interconnection Model (OSI)

Data for messages is also handled at higher levels or "layers" in the computer before transmission. To specify standard protocols for text message handling the International Standards Organization (ISO), in about 1977, proposed a seven-layer model called the *Open Systems Interc*onnection (OSI) model. We give a brief description of the seven OSI layers.

Starting with the bottom layer, the layers are labeled suggestively as follows: (1) Physical, (2) Data link, (3) Network, (4)Transport, (5) Session, (6) Presentation, (7) Application . To simplify the explanation of networking, we shall loosely think of a protocol as a computer process (program) residing in the kernel of an operating system (see Chap. 6) which performs a "service" to a user client who wishes to send a message from a *host* computer (the *source*) to another host (the *destination*) on the Internet. The protocols are arranged in the seven layers approximately to form a *stack*. A protocol interfaces with protocols in the layers above it during receipt of a message in the destination node and interfaces with protocols in the layer below it during sending a message from the source node. At each layer, information pertaining to transmitting the message is adjoined to the message during the sending process and stripped from it during receiving. (We shall elaborate later.) The functionality of the various layers is as follows.

Layer 1 deals with the electrical and physical hardware of the communication medium (e.g. the NIC and cables as in an ethernet network described above). Layer 2 handles data formats of messages, for example as ethernet *frames* and error-correcting codes. It also handles media access control (MAC) and logical link control for the network flow of data in the transmission media as in ethernet cables, for example. Layer 3 deals with the network flow path of message units called *packets,* routing packets through a network from a *source node* to a *destination node*. The particular layer 3 protocol mentioned earlier as IP manipulates Internet addresses of source and destination nodes. Layer 4 has protocols for the transmission of data between nodes on a network flow path. The Internet Protocol (IP) in layer 3 and the Transmission Control Protocol (TCP) in layer 4 together constitute TCP/IP, the layer 4 protocol of choice. It is the main protocol for nearly all Internet transmissions. Higher layer protocols are often ignored. TCP is implemented as a very large C computer program installed on all nodes in the Internet.

It has been implemented to run under Unix and Windows operating systems. In the Unix version, there are about 15,000 lines of source code. (See Wright and Stevens 1995)

How the OSI Layers Work

TCP, by virtue of its program size, is clearly too large to explain fully in the space available here. However, we can give some general idea of how the layer protocols are actuated by an *application program's interface* (API) with the TCP protocol by considering a typical user API for sending a message from a computer node. This will also give some idea of how Internet addressing is organized. The API is programmed by using a set of functions called the *Sockets* system. There are two main Sockets systems, one running under Unix operating systems and the other under Windows. The functions they provide are similar and are discussed below in the section labeled **Sockets**. For now it suffices to know that there are functions for the user to **send** and **receive** messages as sequences of bytes. Of course, there are also functions to specify a source address and a destination address (such a pair of addresses being called a *socke*t).

To send a message from a source computer node to a destination node, the top OSI layer 7 in the source node begins by providing the part of the interface for an application program (API) which generates the raw data for a message. See later discussion of Sockets for other parts of an API, in particular, such information as the source address and the destination address. Layer 7 passes this message information to layer 6, which may convert the raw data from one specified format to another (e.g. *compress* it or *decompress* it). Then layer 5 opens and controls a communication *session* between the two computers regarded as network nodes. (The *session* will be closed when the message is received in the destination node.) In response to a **send** function call, layer 4 provides the actual software for various communication protocols. (There are two main layer 4 protocols, UDP and TCP. We focus on TCP.)

In the layer 4 TCP protocol, it is assumed that each computer node in a network has a unique *IP address*, the first part of which is a network name (assigned by the Network Information Center agency) and the second part of which is a *host name* (of a node) as assigned by a local network administrator. (Networks have these administrative agents.) TCP also provides a *port number* to complete an IP address. The port number can be thought of as designating a *mailbox* register (a memory cell) within the source node which TCP/IP uses to send a message from the node. In the destination address the port number designates a register to receive a message. Each message to be transmitted includes the two IP addresses, source and destination, as fields in its structure. For example, one address format, called IPv4, has a 32-bit address field in a multi-part notation consisting of four bytes (8 bits per byte) separated by dots as in 131.44.2.1, in a dotted-decimal byte value format, denoting the bit string 000011001011000000001000000001.

These four byte fields can be coded to represent a network address, a subnetwork address, and a host address. (See below.) The IP protocol converts a name-form of address to an IP numeric dotted-decimal address format as just illustrated. The name-form is like a familiar email address, for example: "computer-name. university-name.edu" where "edu" is a *domain name.* A TCP/IP "service" program, the Domain Name System, maps name addresses to (numeric) addresses in the dotted-decimal notation. TCP/IP assumes that there are four classes of networks, A, B, C, D. The interpretation of the field for an IP numeric address depends on the network class. As stated, an IP address has two parts: a network part and a host part. In a Class A network, which are very large networks (only 127 of them), there can be 17 million hosts. The first byte field in the dotted-decimal code is the network part and the host part is in the remaining three fields. In a class B network, (about 16,384 of them), there can be 65,000 hosts. Accordingly, the network part of an IP address uses fields 1 and 2 and the host part fields 3 and 4. In a Class C network, (about 2 million in the Internet with addresses using fields 1,2,3), there can only be 254 hosts with addresses in field 4. Next, the data link layer obtains the source MAC address, which is the hardware *media address* of the source NIC and the hardware destination MAC address of the NIC in the API specified destination node computer or router device to which the message is to be forwarded. The message is then sent to a node determined by TCP/IP on the *first hop* of its network path.

To receive the message, TCP/IP works in the computer or the local router in the node determined to be on the first hop. Working up the OSI layers, the first layer hardware receives the message. Next, the data link layer tests if the message MAC address matches the NIC address of the first layer. If not, the message is forwarded. If so, but if the destination IP address network part does not match one of the router's accessible networks, again the message must be forwarded. The router consults its *routing table* (installed by the network administrator) and finds the best path to forward the message on the next hop. For each such hop the message passes from one router to the next which repeats the upward layer protocols. Finally, assuming no errors have occurred, a router is reached for which the destination address is that of a computer on one of its own networks. The message is **received** by being passed up the OSI layers in the destination node, with protocols working to strip out the data part and deliver it to the designated port address.

We have so far omitted to mention that in the source node every message is partitioned into smaller strings of bytes called *packets.* In the source node layer 4, the IP protocol adds Internet routing information to the message and in the data link layer various headings are appended to the packets of data to create the basic units of data transmission called *packets* that are passed finally to layer 1 for transmission. TCP/IP guarantees that a packet is received by the destination node by causing an acknowledging message to be sent back to the source node. If a message is not acknowledged within a certain time, then TCP/IP re-sends the message. The alternative layer 4 protocol, called UDP, does not guarantee receipt of packets, which may still be an acceptable situation. Packets are the units of information transmitted by all protocols. Besides data, packets contain header fields for the destination address and possibly transmission control information. Also packets which are parts of the same message may be disassembled for independent transmission to the destination node where they are reassembled in correct order.

How is a packet transmitted through the Internet? We have just stated that this process involves determining a *route* (a path) and then actually *forwarding* the packet along the route. Routing and forwarding of packets from a source node S can sometimes be done simply by providing a routing table of addresses of possible next *hops* (the nodes immediately connected to node S) that should be used to forward the message to the destination node D. In certain simple cases, the construction of such a table can be based on the destination node address and the graph-theoretic structure of the network in a reasonably sized neighborhood *N* that contains S and D, provided such a neighborhood exists. For long network distances between S and D there are obvious problems. Aside from the unlikely existence of a reasonably sized network neighborhood *N*, the growing number of core networks (in the millions) and hosts per network (hundreds) soon exhausted the available IP address codes for networks. According to the book "Network Processors" by Ran Giladi, Morgan Kaufmann, 2008, many engineering solutions to this problem were considered including, obviously, enlarging the IP address to 128 bits (called IPv6). However, the major engineering solution, called Classless Inter-Domain Routing (CIDR) , added a hierarchical structure of subnets to network addresses and a method of introducing supernet aggregates as well, with a simple address coding notation which could be handled both by hardware protocols at nodes and by software protocols. This complicated the node addressing schemes but resulted in smaller routing tables. One must keep in mind that networks and network technology evolved dynamically and continue to do so in a variety of modes by the many network companies engaged in the engineering of networks; e.g. Bittorrent, Comcast Cable, T-Mobile USA, Time-Warner Cable, VeriSign, Cisco Systems, Hurricane Electric, Netsumo Limited and organizations like the American Registry for Internet Numbers (ARIN), to mention a few listed as involved in the Internet ON 2010 Conference. As stated above earlier, the Internet has a heterogeneous complex graph structure. It does not work perfectly. The OSI model is not perfect but it imposes some order on the protocols, as illustrated above and the huge TCP/IP program helps to make the OSI model work reasonably well. The cited Wright-Stevens book is a detailed account of TCP/IP and the Held book explains some general underlying concepts regarding the routing function. Routing and routers, with their tables installed in kernels by local administrators, are clearly a critical part of the TCP/IP management of message transmission. The tables at nodes in the kernels of the operating systems can be installed by system programs known as daemons coded by local administrators and can be updated to delete network paths which have malfunctioned or to add new paths when new nodes are installed.

Besides its function as a general communication protocol, TCP/IP includes various "well-known" service utilities (e.g. printing) at reserved or "well-known" port numbers in the range 1–1,023. TCP/IP runs under Unix operating systems using Unix system calls. As already noted, the C language code for the Unix TCP/IP program has about 15,000 lines of code. TCP/IP also runs under Windows operating systems. (See Quinn and Shute 1996) Both systems use the Sockets constructs as explained next and in the following Appendix.

Sockets

To understand how protocols work one must keep in mind that the protocols' job is to transmit messages created by application programs. So the application programmer must be able to interface with the OSI layers, in particular, layer 4. An application programming interface (API) between an application program and TCP/IP is provided by a software system known as *Sockets,* which comprises system functions for setting up communication links between programs running on computers in a network. In the OSI model, this API lies between layers 5–7 and layer 4. It consists of functions for programming network applications. (See the following example using JAVA.) There are two main implementations of Sockets, one for Unix operating systems, called Berkeley Sockets (See the book by Wright and Stevens cited above.) and the other for Windows operating systems, called Winsock (See Quinn and Shute cited above.)

The Sockets API is based on a *client-server* model of network communication. Traditionally in systems software, a *server* is a process (program) that receives requests to provide a *service* to another process as *client*, such as requests to print a file. In a network, such requests are in the form of messages sent from a *client* node to the *server* node according to a protocol in the TCP/IP suite of programs. The server may send messages back to the client. Thus, the client-server model provides two-way *communication links* (CL) between processes. For further details on Sockets and the MPI message-passing protocol please see the following Appendix to Computer Networks.

The Larger Contexts of Messages

The Sockets software system is based on a unit of transmitted digital information called a *message*. A *message* is a sequence of data bytes together with headers containing address information to be transmitted by the TCP/IP protocol. But the larger context of a message is treated in the other OSI layers. For example, as noted above, a message can be a single sequence of bytes called a *packet* or a sequence of packets derived from a larger unit of communication such as a video signal, in which case the correlations between successive packets must be maintained by a suitable transmission algorithm. Routing of such inter-dependent packets from a source to a destination node in a network can be a complicated transmission process in modern networks which consist of connections which embody the *convergence* of the technologies of data and traditional tele-communications (phone or video) connections. The optimal transmission of inter-dependent packets over multiple paths is still an open problem. In the following Appendix we consider only the transmission of independent individual messages. Even in this basic case, the choice of an optimum route from source to destination node in the network graph can be a difficult problem for TCP/IP. Note that routing tables need not use optimum routes to a destination.

In some wired networks, the multi-packet transmission problem is somewhat ameliorated by the existence of designated nodes having custom hardware such as routers, switches, hubs or firewalls which perform the task of forwarding messages in a pre-assigned manner. Likewise the problem is more manageable in *managed* wireless networks which have special nodes called *access points* which are pre-set to send messages to other nodes. In other unrestricted networks, called *wireless ad hoc* networks, each node is able to forward messages to other nodes based on the existing constraints of network connectivity; i.e. how busy is a path between two nodes. This is a dynamic network infrastructure involving such communication parameters as delay, jitter, and packet loss. The choice of an "optimum" route requires solving a mathematical optimization problem which may involve multiple multi-hop paths. Multi-path routing of video streams may improve bandwidth and overall video quality, but the distribution of transmission bit rates over possible paths must be determined by an appropriate optimization procedure. This is still an open problem.

Telephone and Data Transmission

At the outset we alluded to analog telecommunications as opposed to digital data transmission, citing the familiar analog example of telephone networks. Voice communication is an important adjunct to computer networking. Indeed, a common method of connecting a computer to the Internet utilizes ordinary telephone connections by means of standard telephone-type twisted-pairs of insulated copper wires. Initially, communication engineers assumed that the rather simple telephone twisted-pair cable could only transmit signals at frequencies in the *baseband* of 300–3,400 Hz (1 Herz(Hz) = 1 cycle per sec), which is adequate for voice service. It was gradually recognized that twisted-pair cables can also transmit signals in the *broadband* range 4KHz–4 MHz. This led to the development of the Digital Subscriber Line (DSL) technology which allows simultaneous transmission of voice and digital data signals on a twisted-pair cable by *multiplexing* them at different frequencies.

A telephone company ISP (e.g. Verizon or ATT) can provide a DSL service channel to a user that allows user baseband telephone voice communication multiplexed with user broadband communication for computer data transmission on the Internet. This requires a simple pluggable installation of a standard *router* device connected to the user's computer network card for router input and the router output connected as input to to an ISP device called a *modem* (modulator-demodulator) which is connected to the service provider's phone jack. The modem receives computer digital data from the router arranged in packets, as described earlier, coded in binary bit patterns and converts the binary sequences into an EM broadband carrier signal *modulated* at two voltage levels representing the 0 and 1 bits. This modulated carrier is transmitted, possibly together with a voice signal, from the user's location over the ISP phone line to an ISP local station (Exchange) whence it is forwarded to the ISP's network connected to the Internet.

Another type of telephone communication that uses the Internet is the VOIP (voice over the Internet) service provided by companies like Skype. In VOIP, ordinary telephone voice signals are transmitted directly over the Internet, that is, without passing through any telephone lines. This requires an analog-to-digital-converter-and-adapter device that samples the analog voice signal coming from the phone, converts the sample voltages to say 8-bit digital codes and arranges these as packets suitably IP-addressed based on the dialed phone number and source phone number. It forwards the packets by connecting to the Internet through an appropriate computer node.

Appendix 1

Sockets

The Sockets API functions are based on the client-server model of interprocess communication in a network. One *process* (e.g. a program on a network node) is designated as a *server*, which receives a request for a "service" from another *process* called a *client*. The request is in the form of a message sent from client to server over the network according to a TCP/IP protocol. The server may send a message back to the client as part of the protocol. Thus, the client-server model provides for two-way communication of messages between processes in a network. If the network is observing the MPI standard for message-passing (as in a *cluster* network described in the HPCC chapter), then since MPI does not assume a client-server model of message-passing, Sockets must impose a virtual client-server model on the processes.

As we shall see, for two computer processes to communicate there must be established a *communication link* (CL) between them. The CL is established by the two processes working in clever collaboration, as we shall now explain. The CL has a *socket* at each of its two ends. A *socket* is just a data structure in each process. The CL consists of the two sockets, the network hardware (e.g. cables and/or EM towers) forming the communication path and some parts of TCP/IP. We can represent this scheme by the following diagram, where the arrows indicate two-way message-passing:

client-node < - - - - - - - -CL- - - - - - - - - > server-node .

Assume the client node has the IP address cli-IP and the server node has the IP address serv-IP, as explained earlier. The client socket is formed by assigning a *client port number*, say cli-port, as part of a socket address in the client node. Then the address of the client socket is assigned to a variable, say cli-sock-addr, so that

cli-sock-addr = (cli-IP, cli-port).

Similarly, for the server node the socket address is assigned, so that

serv-sock-addr = (serv-IP, serv-port) .

For programming purposes the sockets must have names, say cli-sock-id and serv-sock-id. The pair of *socket data structures* can then be represented as

client_socket = (cli-sock-id, cli-sock-addr, serv-sock-addr)
server_socket = (serv-sock-id, serv-sock-addr, cli-sock-addr).

The Sockets API provides functions that the two processes can use to create such data structures in their own nodes, as we shall show below. Once such a pair has been created, the two processes are peers as far as message passing is concerned. The virtual client-server relation used to create the structures can then be ignored. Each process can receive/send messages from/to the other over the CL by referring to its own socket name and the associated data structure. To see this we first describe the main Socket operations for doing the send and receive message functions.

Socket Send/Receive Functions

In Unix Sockets a process can send a message by calling one of the functions **write, sendto, or sendmsg**. Winsock provides the **send** function. We shall use the simpler Winsock syntax **send** to explain the general idea of sending a message by means of Sockets. Likewise, we shall use the Winsock syntax **recv** for receiving a message.

In Sockets, messages consist of bytes in a buffer array. Let bufc be such an array in process X. (Recall that the context from which bufc is constructed is not part of the Sockets system.) Suppose X wants to send bufc to process Y over a link CL which has been created with X as the client and Y as the server. (See below.) The socket for X in this CL is named cli-sock-id. The **send** function has the following header:

int send(int cli-sock-id, char *bufc, int lngthbufc, int n) .

The parameter lngthbufc is the size of bufc in bytes. n indicates certain options which we ignore for the moment and use 0 as the default for a normal **send**. Recall that the * symbol in C++ denotes a pointer to an array. The function call statement

send(cli-sock-id, *bufc, lngthbufc, 0)

when executed in X causes TCP/IP to assemble an appropriate *packet* (or packets) of bytes from bufc and pass it to protocol IP to be sent to serv-IP given in serv-sock-addr in the socket data structure cli-sock-id above. serv-IP is the node where Y is located. In X, the packets may be partly disassembled and then delivered to TCP at serv-port where the message is extracted and placed in a temporary buffer buftemp. To receive the message, Y must execute a **recv** call. The header for the recv function is

int recv(int serv-sock-id, *bufs, int lngthbufs, int n) .

The call

recv(serv-sock-id, *bufs, lngthbufs, 0)

executed in Y interacts with TCP at serv-port. If a message has already been delivered to serv-port, it will be transferred from buftemp to bufs and process Y can proceed with its own execution. Otherwise, Y waits at the recv call until a message is sent by X. After **recv** executes, TCP sends an acknowledge message to cli-node at cli-port as obtained from serv-sock-id. TCP in cli-node interacts with cli-port to obtain the acknowledgment and the **send** call is completed. Process X can proceed with its own execution.

The send-receive relation between X and Y is symmetric. Clearly, Y can send a message to X using the same pair of sockets. The client-server aspects are ignored. The procedure outlined above works with “cli” and “serv” interchanged.

Blocking and Non-blocking Sockets

The sequential logic of the above send-receive procedure poses various questions about the synchronization of the steps relative to the execution of X and Y. For example, acknowledgment of receipt of a message could be done at the point when the message is delivered to buftemp without waiting for a **recv** call by Y. In that case, X need not be *blocked* at the send call waiting for Y to execute a **recv**. Another possibility is to allow X to proceed with its execution as soon as its TCP has transferred the message out of bufc, say to a temporary system buffer. This would allow execution of X to partly overlap with the remaining communication steps. Likewise, Y need not be *blocked* at its **recv** call waiting for X to do a **send** call, but instead Y can be allowed to proceed after some error is returned to the **recv** call. Blocking raises the possibility of *deadlocks* in process execution, as for example if X sends to Y while Y sends to X with no intervening recv calls. In this situation, both X and Y would be blocked and be *deadlocked*. One way to eliminate deadlocks is to have an automatic *timeout* set by TCP which would limit the time a blocked call waits. In Winsock, **send** has an automatic timeout set by TCP. For **recv** the application programmer can set timeouts. Rather than use timeouts, we consider the choice, provided in Sockets, of declaring a socket as being *blocking* or *non-blocking* when it is created by a call to the function **socket**. (See below.) By default, a socket is blocking when it is created. A socket in Unix can be made non-blocking by a call to **fcntl**. In that case, the send and receive calls return whether or not the message passing steps are completed and an error message indicates either success or the current socket state. An implementation of MPI (see below) can pass these error messages on to the programmer for error handling.

Creating a Client-Server Socket Pair under MPI Implementations

Assume that the MPI standard has been implemented on a network of nodes. This means that a library of MPI functions (see below) is available for the network programmer. For simplicity of explanation, we assume that in an application there is exactly one process per node. The application programmer organizes the application into a set of quasi-independent processes which can execute concurrently and independently until they reach points where messages must be sent/received to-from each other. (See the HPCC chapter on clusters.) Under MPI, to identify processes each process is assigned an integer *rank*. Its assigned rank can be retrieved by a process by an MPI function call. There is also an application *configuration file* provided by the programmer defining the desired interprocess connection topology. Suppose an application program requires that the process of rank i needs to communicate with the process of rank j. Suppose as a rule that i $<$ j is taken to mean that process i is the client and process j the server. (Other rules are possible.) The following steps will set up a pair of socket structures on a link CL connecting processes i and j for two-way message-passing.

Step 1. In the client and server nodes the respective processes make the following respective calls to the system **socket** function

cli-sock-id = **socket** (AF-INET, SOCK-STREAM, 0)
serv-sock-id = **socket**(AF-INET, SOCK-STREAM, 0).

Here AF-INET is the internet address family which conforms to the TCP/IP protocol for node addresses and port numbers, SOCK-STREAM is the TCP reliable message-passing protocol and the 0 tells the process to use the TCP protocol. As shown, these calls return a proper socket name for programming.

In the client-server model, a server can have many clients. Therefore, the steps for creating a server socket are different from those for a client. The Sockets library provides functions for these steps. We begin with the server.

Step 2a. The **bind** function and serv-sock-addr.
Having created the socket name serv-sock-id for the server socket in step 1, the server process (rank j by our rule) calls the Sockets **bind** function to bind a socket address serv-sock-addr to serv-sock-id. This involves somewhat complex declarations using the predefined Sockets struct *sockaddr_in* which is given in two files sys/sockets.h and sys/types.h which must be included in the MPI implementation. This C++ struct has the following format (with comments /*. . .*/):

```
struct sockaddr_in{
   short          sin_family       /*address family*/
  u_short         sin_port          /*16 bit port number*/
struct   in_addr  sin_addr         /*32 bit node IP address*/
     char         sin_zero(8) }     /* internal system use*/
The struct in_addr is predefined in the file netinet/in.h  .
```

The address serv-sock-addr is calculated by the following C++ declarations and assignments using this struct:

```
struct  sock-addr_in   serv-sock-addr; /*declare serv-sock-addr  to be such a struct*/
      int serv_port;                              /* declare serv_port*/
      int serv-sock-id;
      serv-port = port(j, i);
      bzero((char*)6 serv-sock-addr   sizeof (serv-sock-addr));  /* initialize*/
      serv-socck-addr.sin_family = AFNET;
      serv-sock-addr.sin_port  = htonl(serv-port);        /*server port number*/
      serv-sock-addr. sin_addr = htonl(INADDR_ANY);  /*server node address*/.
```

Here **port**(j,i) is the implementer's function to compute a port number for serv-sock-addr. htonl is a Sockets function that converts it to a compatible network format. The constant INADDR_ANY is a wild card that allows Sockets to select an IP address from a cluster system file. The call to **bind** is is then

bind(serv-sock-id , (struct serv-sock-addr*), &serv-sock-addr, sizeof(serv-sock-addr));

The **bind** returns 0 on success and SOCKET_ERROR on failure (as, for example, when another process has already bound to this socket address).

Step 2b. The Server **Listens** and **Accepts** Calls.
In step2a, a server socket address serv-sock-addr is associated with serv-sock-id. A server socket must be created for each client that needs to communicate with this server. Each server socket has the same serv-sock-addr but a different cli-sock-addr. Sockets provides two functions, **listen** and **accept**, to create multiple client links to a server. Note that a node having multiple communication links to other nodes must be set up with a server socket as follows.

The **listen** call causes the server to queue up to five client **connect** calls (see below) in a certain time interval. A loop of accept calls then completes each connect call in the queue by creating a new server socket for each client. All clients send their connect calls to the same address serv-sock-addr. TCP delivers all connect calls to the same server port. It also delivers the client port number so that the server socket can be created. The header for the **listen** function is as follows:

int **listen**(int serv-sock-id, int quelength).

serv-sock-id is the is the socket id in the **bind** call in step2a above and quelength is an integer between 1 and 5 establishing a queue of that length while the server receives **connect** calls. The **listen** calls returns 0 on success and SOCKET_ERROR on failure.

An **accept** call following the **listen** call blocks until a client **connect** call is made to serv-sock-id. The accept function has the following header:

int **accept** (int serv-sock-id, struct sock-addr_in *client, int *addrlength).

where serv-sock-id is as above, and the other parameters are OUT parameters for a client address and address length. A **connect** call (see below) to the server causes

TCP to send a packet from a client node to the port address in serv-sock-addr in the server node. The packet contains the client address cli-sock-addr. This is extracted by an **accept** function call and placed in the socket data structure of a new server socket for that client. cli-sock-addr is also stored in the struct pointed to by *client for possible other server use.

Step 3. The Client **Connect** Call
In a client node, the client socket in a link to the server is created by a **connect** call. The Sockets **connect** function has the following header:

int **connect**(int cli-sock-id, struct sock-addr_in *servaddr_in, int namelength).

Here cli-sock-id is created in the **socket** call in the client as above. *servaddr_in is a pointer to a server address struct which must be initialized by the client using the same calculations of sin_port and sin_addr as in the server above. A **connect** call in a client node causes a client address (IP address of client node and a port number) to be bound to cli-sock-id, creating the client socket in the client node. Then a message containing the client address is sent to the server as a request for action by an **accept** call. As explained above, the **accept** will extract the client socket address and create the server socket data structure in the server. This completes the connection link between the two processes. **Connect** calls can be set to be nonblocking, in which case they return an error if not completed by an **accept**. They can also be repeated in a loop until accepted.

MPI Collective Communication

Suppose there are p processes organized as a group in an application program. (See Chap. 7 on HPCC clusters.) In many applications, a process must send its partial results to all p processes or receive messages from all p processes in a group. This is called *collective communication* under the MPI standard. To simplify programming, MPI provides for various collective functions which carry out such collective communications with a single call, which is then automatically implemented by multiple send's and recv's by the p processes. A collective function call must be made by all p processes involved. Here are some examples provided in MPI implementations.

Broadcast. A source process sends the same message to the other p-1 processes,

Gather. A destination process receives a message from each of the other p-1 processes and concatenates the messages in a buffer in rank order.

AllGather. A multiple Gather in which all processes are treated as destinations and each ends up with the same concatenated message.

The headers are as follows.

int **Broadcast** (char *buffer, int bufferlength, int source).

Each process must issue the same call statement specifying the same source node and buffer.

int **AllGather**(char *out_buffer, int out_buffer_length, int *in_buffer).

out_buffer is the address of the buffer in each process of the message sent by that process to all processes. in_buffer is the address of the buffer in each process for receiving and concatenating the messages sent by all processes. The length of in-buffer must be at least px(out_bffer_length).

For other collective message functions see Snir et al. (1996) below.

Appendix 2

Graph theory

This Appendix r is based in part on the Noether Lecture given by Chung (2009)

In this chapter on Computer Networks we mentioned some applications of mathematical methods to the construction of algorithms used in transmission of information in computer networks. One of the main sources of such methods is the subject called *graph theory*, which we now summarize. A graph, G(V, E), (defined above in the Introduction to Computer Networks as a means to represent and study networks) can be viewed literally as a set of points V called *nodes* or *vertices* and a set of arcs E called *edges* connecting certain pairs of vertices. More abstractly, a graph defines a binary relation on a set V. In a computer network, such as the Internet or a part of it, a graph representing the network has vertices which represent individual signal transmission devices such as computers and cell phones, locations of such devices called *web sites,* and transmitter relay towers which forward messages. The edges represent wired or wireless connections between pairs of vertices. Transmission of messages between vertices can be analyzed in terms of connections in a graph representing the network. Obviously, the vertices representing transmission towers will connect to multiple edges (i.e. have high *degree as* graph vertices) connecting to neighbor vertices which are devices which originate messages. Likewise, webpage sites will be vertices of high *degree* if they are popular resources, since other devices will access them by interlink commands. These network operational factors impose some structure on their graphs. To comprehend these structural features we can apply graph theory.

Graph theory deals with combinatorial and geometric problems that arise in analyzing properties of a graph G(V, E), such as the existence of *paths* consisting of chains of adjacent edges, *connectivity* of two vertices by paths, shortest paths connecting two vertices, and other path properties. These combinatorial/geometric problems are obviously relevant to computer networks represented as graphs. In the past decade, graph theory has gone through a remarkable transformation. The change is in large part due to the huge number of data sets that we are confronted with in modern computer networks with their numerous webpages. A main way to sort through numerous massive data sets is to build and examine an abstract network formed by intrinsic interrelations between the data sets. For example, Google's successful WWW search algorithms are based on a WWW graph which contains all Webpages as vertices and hyperlinks as edges. (See Introduction.) The search for a particular piece of information is based on the relations between data sets relevant to that piece of information. To appreciate the scope of the search problem one must be aware of the diversity of data sets which exist at various websites. There are now all sorts of information networks such as biological networks built from biological databases and social networks formed by email (e.g. Facebook), phone calls, instant messaging, etc., as well as various types of physical networks which span the earth. Graph theory can be used to comprehend and analyze the functioning of these networks.

Graph theory has 200 years of history studying the mathematical structures G(V, E) called *graphs*. In the past, graph theory has been used in a wide range of areas. However, never before have we been confronted by graphs of not only tremendous sizes (number of vertices) but also extraordinary richness and complexity (of edge configurations) both at a theoretical and a practical level. Numerous challenging problems have attracted the attention and imagination of researchers from physics, computer science, engineering, biology, social science and mathematics. A new area of "network science" has emerged, calling for a sound scientific foundation and rigorous analysis of networks for which graph theory is ideally suited. These real-world networks and their associated graphs are massive and complex but illustrate amazing coherence. Empirically, most "real-world" graphs have the following properties:

- Sparsity – The number of edges is within a constant multiple of the number of vertices.
- "Small world phenomenon" – Any two vertices are connected by a short path. Two vertices having a common neighbor are more likely to be neighbors (A *neighbor* of a vertex, v, is a vertex connected to v by an edge).
- Power law degree distribution – The *degree* of a vertex is the number of its neighbors. The number of vertices with degree j is proportional to $j^{-\beta}$ for some fixed positive constant β.

In dealing with graphs representing such networks, many basic questions arise: What are basic structures of such large networks? How do they *evolve* from smaller networks? (In the real world networks begin small and then grow as they are used; e.g. more cell phones add new vertices). What are the underlying principles that dictate their communication behavior? How are *subgraphs* related to a large (and often *incomplete*) *host* graph? What are the main graph invariants that capture the properties of such large graphs?

To answer some of these questions, we shall first delve into the wealth of knowledge from the past although it is often not enough. In the past 30 years, there has been a great deal of progress in *combinatorial* and *probabilistic* methods as well as *spectral* methods. However, traditional probabilistic methods mostly consider the same probability distributions for all vertices or edges while real graphs have non-uniform and clustered distributions of edges. The classical algebraic and analytic methods are efficient in dealing with highly symmetric structures while real-world graphs are quite the opposite. Guided by examples of real-world graphs, we are compelled to improvise, extend and create new theory and methods. Here we will discuss new results and ideas in several topics in graph theory which are rapidly developing. The topics include ***random graph theory*** for any given degree distribution, ***percolation*** in general host graphs, ***PageRank*** for representing quantitative correlations among vertices and the **game theory** aspects of graphs.

Some Basics of Graph Theory

Before proceeding with these topics, it may be convenient to the reader if we review at this point some basic aspects of graph theory. Graphs G(V, E) are often depicted by diagrams consisting of points denoting the vertices in V and arcs drawn between certain pairs of points denoting the edges in E. An arc may have an arrowhead, in which case the edge is *directed*, indicating an *ordered pair* of vertices. While providing an intuitive visual geometric picture of a graph, such diagrams become somewhat less visually apprehended when the graph size |V| (number of vertices in V) is large. Therefore, we shall resort to other than visual representations by diagrams and introduce certain matrix representations of graphs G(V, E) that arise naturally. Let V have n vertices, i = 1,. . .,n.

The **incidence** matrix I is an array of |V| rows and |E| columns where the entry I(i, j) is 1 if vertex i is an endpoint of edge j and 0 otherwise;

The **adjacency** matrix A is an nXn matrix where entry A(i, j) = 1 if there is an edge from vertex i to vertex j and otherwise 0. (More generally A(i, j) = q if there are q edges joining vertex i to vertex j. e.g. a roadmap can have two cities connected by q roads.)

The **Laplacian** (or **admittance** or **Kirchoff**) matrix is the matrix D – A, where D is a diagonal matrix having the degree of vertex i in element d_{ii}. The normalized Laplacian is the matrix $I - D^{1/2} AD^{1/2}$.

The **distance** matrix (d(i, j)) has d(i, j) equal to the *length* of the shortest path connecting vertex i to vertex j., where length of a path is the number of edges (*hops*). If there is no path, the distance is infinite. It is a simple exercise to prove that d(i,j) can be derived from powers of A:

namely, d(i, j) = min (m such that A^m (i, j) is nonzero).

Hint: the (i, ,j) element b(i,j) in A^2 is given by $b(i, j) = \sum_k A(i, k) A(k, j)$ which is non zero if there is a path of length 2 from i to j passing through some vertex k. Then use induction on m where $A^m = A\ A^{m-1}$.

A different notion of distance applies to a graph *labeled* by numerical weights assigned to the edges to represent various geometric properties such as actual distances between vertices as in a roadmap or a computer network. Consider such a labeled graph G having non-negative weights. It is of interest to find the shortest weighted path between vertices. E. Dijkstra in 1959 published an algorithm to find such shortest paths. This algorithm finds the shortest path from a vertex v in G to any other vertex in G. It is typical of purely computer science algorithms, involving a clever search strategy.

Random Graph Theory for General Degree Distributions

The primary subject in the study of random graph theory is the *classical* random graph G(n, p), introduced by Erdős and Rényi in 1959 (Erdős and Rényi 1959, 1960) (also independently by Gilbert (1959)). In G(n, p), each pair in a set of n vertices is chosen at random to be an edge with probability p. So for p = ½ say, about half of the n(n – 1)/2 pairs of vertices are joined by edges chosen "at random". Thus in the graphs G(n, p) the set E of edges can be regarded as having a random-looking geometric structure determined only by p, as opposed to a graph G(V, E) in which E has a well-determined regular-looking structure, say making the set V fully connected (e.g. *complete*) or decomposable into a few connected *components*. In a series of papers, Erdős and Rényi gave an elegant and comprehensive analysis describing the formation of E (i.e. the *evolution* of G(n, p)) as p increases. In real-world network graphs, the network evolves say by adding edges as more network components come on line. It seems clear that a random graph G(n, p) must have the same expected degree at every vertex. (e.g. Consider how p = ½ restricts the creation of edges and thus equalizes the average degree at each vertex.) Therefore, G(n, p) does not capture some of the main behaviors of real-world graphs which, as suggested above, usually have different degrees at different vertices. In the WWW graph, at some vertices designating popular websites the degree (number of neighbors) would be much higher than at unpopular websites. Nevertheless, the approaches and methods in the classical random graph theory of G(n, p) provide the foundation for the study of non-classical random graphs with general degree distributions. We will present some classical random graph theory.

Many random graph models have been proposed in the study of information networks graphs but there are basically two different models. The "on-line" model mimics the real-world growth or decay of a dynamically changing network and the "off-line" model of random graphs consists of families of graphs with some specified edge probability distributions.

One on-line model is based on the *preferential attachment* scheme which can be described as "the rich get richer". The preferential attachment scheme has been receiving much attention in the recent study of complex networks (Barabási and Albert 1999; Mitzenmacher 2004) but its history can be traced back to Vilfredo Pareto in 1896, among others. At each tick of the clock (so to speak), a new edge is added so that each of its endpoints is chosen with probability proportional to their degrees. The higher the degrees, the more likely is an edge added. It can be proved

(Bollabás and Riordan 2003; Chung and Lu 2006; Mitzenmacher 2004) that the preferential attachment scheme leads to a power law degree distribution. There are several other on-line models including the duplication model (which seems to be more feasible for biological networks, see Chung et al. 2003c).

There are two main on-line graph models for graphs with general degree distribution – the *configuration model* and random graphs with expected degree sequences. A random graph in the configuration model with degree sequence at the n vertices d_1, $d_2, \ldots, d_n$ is defined by choosing a random matching on $\sum d_i$ "pseudo nodes" where the pseudo nodes are partitioned into parts of sizes d_i, for i =1, . . ., n. Each part is associated with a vertex. By using results of Molloy and Reed (1995, 1998), it can be shown (Aiello et al. 2000) that under some mild conditions, a random power law graph with exponent β almost surely has no giant component if $\beta \geq \beta_0$ where β_0 is a solution to the equation involving the Riemann zeta function

$$\zeta(\beta - 2) - 2\,\zeta(\beta - 1) = 0.$$

The general random graph model G(w) with expected degree sequence $w = (w_1, w_2, \ldots, w_n)$ follows the spirit of the Erdős-Rényi model. The probability of having an edge between the ith and jth vertices is defined to be $w_i w_j/\mathrm{Vol}\,(G)$ where Vol (G) denotes $\sum w_i$. Furthermore, in G(w) each edge is chosen independently of the others and therefore the analysis can be feasibly carried out. It was proved in Chung and Lu (2002b), that if the expected average degree is strictly greater than 1 in a random graph in G(w), then there is a giant component (i.e., a connected component of volume a positive fraction of that of the whole graph). Furthermore, the giant component almost surely has volume $\delta\mathrm{Vol}\,(G) + O(\sqrt{n}\,\log^{3.5} n)$. where δ is the unique nonzero root of the following equation (Chung and Lu 2006):

$$\sum w_i e^{-wi\delta} = (1 - \delta) \sum w_i. \tag{7.1}$$

Because of the robustness of the G(w) model, many metric properties can be derived. For example, a random graph in G(w) has average distance almost surely equal to (1 + o(1)) logn/logw* where $w^* = \sum w_i^2 / \sum w_i$ and the diameter is almost surely Θ(log n/ log w*) provided some mild conditions on w are satisfied (Chung and Lu 2002a). For the range $2 < \beta < 3$ where the power law exponents β for numerous real networks reside, the power law graph can be roughly depicted as an "octopus" with a dense subgraph having small diameter O(log log n), as the core, while the overall diameter is O(log n) and the average distance is O(log log n) (see Chung and Lu 2006).

For the spectra of power law graphs, there are basically two competing approaches. One is to prove analogues of Wigner's semi-circle law (which is the case for G(n, p)) while the other predicts that the eigenvalues follow a power law distribution (Faloutsos et al. 1999). Although the semi-circle law and the power law have very different descriptions, both assertions are essentially correct if the appropriate matrices associated with a graph are considered (Chung et al. 2003a, b). For $\beta > 2.5$, the largest eigenvalue of the adjacency matrix of a random power law graph is almost surely (1 + o(1)) √ m where m is the maximum degree. Moreover, the k largest eigenvalues have power law distribution with exponent 2β −1 if the maximum degree is sufficiently large and k is bounded above by a function depending on β, m and w. When $2 < \beta < 2.5$, the largest eigenvalue is heavily concentrated at $cm^{3-\beta}$ for some constant c depending on β and the average degree. Furthermore, the eigenvalues of the (normalized) Laplacian satisfy the semi-circle law under the condition that the minimum expected degree is relatively large (Chung et al. 2003b). The one-line model is obviously much harder to analyze than the on-line model. One possible approach is to couple the on-line model with the off-line model of random graphs with a similar degree distribution. This means to find the appropriate conditions under which the on-line model can be sandwiched by two off-line models within some error bounds. In such cases, we can apply the techniques from the off-line model to predict the behavior of the on-line model (see Chung and Lu 2004).

Random Subgraphs in a Given Host Graphs

Almost all information networks that we observe are subgraphs of some host graphs that often have sizes prohibitively large or with incomplete information. A natural question is to attempt to deduce the properties of a random subgraph from the host graph and vice versa. It is of interest to understand the connections between a graph and its subgraph. What invariants of the host graph can or cannot be translated to its subgraph? Under what conditions, can we predict the behavior of all or any subgraphs? Can a sparse subgraph have very different behavior from its host graph? Here we discuss some of the work in this direction. Many information networks or social networks have very small diameters (in the range of log n), as dictated by the so-called "small world phenomenon". However, in a recent paper by Liben-Nowell and Kleinberg (2008) it was observed that the tree-like subgraphs derived from some chain-letter data seem to have relatively large diameter. In the study of the Erdős-Rényi graph model G(n, p), it was shown (Rényi and Szekeres 1967) that the diameter of a random spanning tree is of order √n, in contrast with the fact that the diameter of the host graph K_n is 1. Aldous (1990) proved that in a regular graph G with a certain spectral bound σ, the expected diameter of a random spanning tree T of G, denoted by diam(T) has expected value satisfying

$$c\ \sigma\sqrt{n}/\log n \le E(\mathrm{diam}(T)) \le c'\sqrt{n}\ \log n/\sqrt{\ }\ \sigma$$

for some absolute constant c. In (Chung et al.), it was shown that for a general host graph G, with high probability the diameter of a random spanning tree of G is between c √n and c′√n log n, where c and c• depend on the spectral gap of G and the ratio of the moments of the degree sequence.

One way to treat random subgraphs of a given graph G is as a (bond) percolation problem. For a positive value $p \leq 1$, we consider G_p which is formed by percolation, retaining each edge independently with probability p, and discarding the edge with probability $1 - p$. A fundamental problem of interest is to determine the critical probability p for which G_p contains a giant connected component. In the applications of epidemics, we consider a general host graph being a contact graph, consisting of edges formed by pairs of people with possible contact. The question of determining the critical probability then corresponds to the problem of finding the epidemic threshold for the spreading of the disease. Percolation problems have long been studied (Grimmett 1989; Kesten 1982) in theoretical physics, especially with the host graph being the lattice graph Z^k . Percolation problems on lattices are known to be notoriously difficult even for low dimensions and has only been resolved very recently by bootstrap percolation (Balogh et al.; Balogh et al.).

In the past, percolation problems have been examined for a number of special host graphs. Ajtai, Komlos and Szemerédi considered the percolation on hypercubes (Ajtai et al. 1982). Their work was further extended to Cayley graphs (Borgs et al. 2005a, b; Borgs et al. 2006; Malon and Pak 2002) and regular graphs (Frieze et al. 2004). For expander graphs with degrees bounded by Alon et al. (2004) proved that the percolation threshold is greater than or equal to 1/(2d). In the other direction, Bollobás et al. showed that for dense graphs (where the degrees are of order $\Theta(n)$), the giant component threshold is $1/\rho$ where ρ is the largest eigenvalue of the adjacency matrix. The special case of having the complete graph K_n as the host graph concerns the Erdős-Rényi graph G(n, p) which is known to have the critical probability at 1/n as well as the "double jump" near the threshhold. For general host graphs, the answer has been elusive. One way to address such questions is to search for appropriate conditions on the host graph so that percolations can be controlled. Recently it has been shown (Chung et al. 2009) that if a given host graph G satisfies some (mild) conditions depending on its spectral gap and higher moments of its degree sequence, for any $\varepsilon > 0$, if $p > (1 + \varepsilon)/d^*$ then asymptotically almost surely the percolated subgraph G_p has a giant component. In the other direction, if $p < (1 - \varepsilon)/d^*$then almost surely the percolated subgraph G_p contains no giant component. We note that the second order average degree is $d^* = \sum {d_v}^2 / \sum d_v$ where d_v denotes the degree of v.

In general, subgraphs can have spectral gaps very different from that of the host graph. However, if a graph G has all its nontrivial eigenvalues of the (normalized) Laplacian lying in the range within σ from the value 1, then it can be shown (Chung and Horn 2007) that almost surely a random subgraph G_p has all its nontrivial eigenvalues in the same range (up to a lower order term) if the degrees are not too small.

PageRank and Local Partitioning

In graph theory there are many essentially geometrical notions, such as *distances* (typically, the number of hops required to reach one vertex from another), *cuts* (i.e., subsets of vertices/edges that separate a part of the graph from the rest), *flows* (i.e., combinations of paths for routing between given vertices), and so on. However, real-world graphs exhibit the "small world phenomenon", so any pair of vertices are connected through a very short path. Therefore the usual notion of graph distance is no longer very useful. Instead, we need a quantitative and precise formulation to differentiate among nodes that are 'local' from 'global' and 'akin' from 'dissimilar'. This is exactly what *PageRank* is meant to achieve. In 1998, Brin and Page (1998) introduced the notion of PageRank for Google's Web search algorithm. Different from the usual methods of pattern matching previously used in data retrieval, the novelty of PageRank relies entirely on the underlying Webgraph to determine the 'importance' of a Webpage. Although PageRank is originally designed for the Webgraph, the concept and definitions work well for any graph. Indeed, PageRank has become a valuable tool for examining the correlations of pairs of vertices (or pairs of subsets) in any given graph and hence leads to many applications in graph theory.

The starting point of the PageRank is a typical random walk on a graph G with edge weights w_{uv} for edges (u, v). The *probability transition m*atrix P is defined by: $P(u, v) = w_{uv}/d_u$ where $d_u = \sum_v w_{uv}$. For a *preference* (or *seed*) vector s, and a *jumping constant* $\alpha > 0$, the PageRank, denoted by $\mathrm{pr}(\alpha, s)$ as a row vector, can be expressed as a series of random walks as follows:

$$\mathrm{pr}(\alpha, s) = \alpha\sum_{k}(1 - \alpha)^{k} sP^{k}. \tag{7.2}$$

Equivalently, $\mathrm{pr}(\alpha, s)$ satisfies the following recurrence relation:

$$\mathrm{pr}(\alpha, s) = \alpha s + (1 - \alpha)\mathrm{pr}(\alpha, s)P. \tag{7.3}$$

In the original definition of Brin and Page (1998), s is taken to be the constant function with value 1/n at every vertex, motivated by modeling the behavior of a typical web surfer who moves to a random page with probability α and clicks a linked page with probability $1 - \alpha$. Because of the close connection of PageRank with random walks, there are very efficient and robust algorithms for computing and approximating PageRank (Andersen et al.; Berkhin; Jeh and Widom 2003) This leads to numerous applications including the basic problem of finding a 'good' cut in a graph. A quantitative measure for the 'goodness' of a cut that separates a subset S of vertices is the Cheeger ratio:

$$h(S) = |E(S, S^{-})|/\mathrm{vol}(S)$$

where $E(S, \bar{S})$ denotes the set of edges leaving S and vol (S) = dv. The Cheeger constant h_G of a graph is the minimum Cheeger ratio over all subsets S with vol (S) $\leq$ vol (G)/2. The traditional divide-and-conquer strategy in algorithmic design relies on finding a cut with small Cheeger ratio. Since the problem of finding any cut that achieves the Cheeger constant of G is NP-hard (Garey and Johnson 1979), one of the most widely used approximation algorithms was a spectral partitioning algorithm. By using eigenvectors to line up the vertices, the spectral partitioning algorithm reduces the number of cuts under consideration from an exponential number of possibilities to a linear number of choices. Nevertheless, there is still a performance guarantee provided by the Cheeger inequality:

$$2\mathrm{h}_\mathrm{G} \geq \lambda \geq \mathrm{h}_\mathrm{f}^2/2 \geq \mathrm{h}_\mathrm{G}^2/2$$

where h_f is the minimum Cheeger ratio among subsets which are initial segments in the order determined by the eigenvector f associated with the spectral gap λ. For large graphs with billions of nodes, it is not feasible to compute eigenvectors**.** In addition, it is of interest to have local cuts in the sense that for given seeds and the specified size for the parts to be separated, it is desirable to find a cut near the seeds separating a subset of the desired size. Furthermore, the cost/complexity of finding such a cut should be proportional to the specified size of the separated part but independent of the total size of the whole graph. Here, PageRank comes into play. Earlier, Spielman and Teng (2004) introduced local partitioning algorithms by using random walks with the performance analysis using a mixing result of Lovasz and Simonovitz (1993) (also see Mihail 1989). As it turns out, improved by using PageRank instead of random walks, there is **an** partitioning algorithm (Andersen et al.) for which the performance is supported by a local Cheeger inequality for a subset S of vertices in a graph G:

$$\mathrm{h}_\mathrm{S} \geq \lambda_\mathrm{S} \geq \mathrm{h}_\mathrm{g}^2/8\log\mathrm{vol}(\mathrm{S}) \geq \mathrm{h}_\mathrm{S}^2/8\log\mathrm{vol}(\mathrm{S})$$

where λ_S is the Dirichlet eigenvalue of the induced subgraph on S, h_S is the local Cheeger constant of S defined by $\mathrm{h}_\mathrm{S} = \min\,\{\mathrm{h}(\mathrm{T})\text{: } \mathrm{T} \subseteq \mathrm{S}\}$ and h_g is the minimum Cheeger ratio over all PageRank g with the seed as vertices in S and α appropriately chosen depending only on the volume of S. This approximation partition algorithm can be further improved using the fact that the set of seeds for which the PageRank leads to the Cheeger ratio satisfying the above local Cheeger inequality is quite large (about half of the volume of S). We note that the local partitioning algorithm can also be used as a subroutine for finding balanced cuts for the whole graph. Note that PageRank is expressed as a geometric sum of random walks in (7.2). Instead we can consider an exponential sum of random walks, called heat kernel pagerank, which in turn satisfies the heat equation. The heat kernel pagerank leads to an improved local Cheeger inequality (Chung 2007, 2009) by removing the logarithmic factor in the lower bound. Numerous problems in graph theory can possibly take advantage of PageRank and its variations, and the full implications of these ideas remain to be explored.

Network Games

In morning traffic, every commuter chooses his/her most convenient way to get to work without paying attention to the consequences of the decision to others. The Internet network can be viewed as a similar macrocosm which functions neither by the control of a central authority nor by coordinated rules. The basic motivation for each individual can only be deduced by greed and selfishness. Every player chooses the most convenient route and use strategies to maximize possible payoff. In other words, we face a combination of game theory and graph theory for dealing with large networks both in quantitative analysis and algorithm design. Many questions arise. Instead of the existence of Nash equilibrium, how can we compute and how fast does it converge to the equilibrium? There has been a great deal of progress in the computational complexity of Nash equilibrium (Chen and Deng 2006; Daskalakis et al.). The analysis of selfish routing comes naturally in network management. How much does uncoordinated routing affect the performance of the network, such as stability, congestion and delay? What are the tradeoffs for some limited regulation? The so-called "price of anarchy" refers to the worst case analysis to evaluate the loss of collective welfare from selfish routing. There has been extensive research done on selfish routing (Roughgarden and Tardos 2002). The reader is referred to several surveys (Feldmann et al. 2003; Kontogiannis and Spirakis 2005) and some recent books on this topic (Roughgarden 2006).

Many classical problems in graph theory can be re-examined from the perspective of game theory. One popular topic on graphs is chromatic graph theory. For a given graph G, what is the minimum number of colors needed to color the vertices of G so that adjacent vertices have different colors? In addition to theoretical interests, the graph coloring problem has numerous applications in the setting of conflict resolution. For example, each faculty member (as a vertex) wishes to schedule classes in a limited number of classrooms (as colors). Two faculty members who have classes with overlapping time are connected by an edge and then the problem of classroom scheduling can be viewed as a graph coloring problem. Instead of having a central agency to make assignments, we can imagine a game-theoretic scenario that the faculty members coordinate among themselves to decide a non-conflicting assignment. Suppose there is a payoff of 1 unit for each player (vertex) if its color is different from all its neighbors. A proper coloring is then a Nash equilibrium since no player has an incentive to change his/her stragegy. Kearns et al. (2006) conducted an experimental study of several coloring games on specified networks. Many examples were given to illustrate the difficulties in analyzing the dynamics of large networks in which each node takes simple but selfish steps. This calls for rigorous analysis, especially along the line of the combinatorial probabilistic methods and generalized Martingale approaches that have been developed in the past 10 years (Chaudhuri et al. 2008). Some work in this direction has been done on a multiple round model of graph coloring games (Chaudhuri et al. 2008) but more work is needed.

Summary

It is clear that we are at the beginning of a new journey in graph theory, emerging as a central part of the information revolution. It is a long way from the "seven bridges of K¨onigsberg", a problem asked by Leonhard Euler in 1736. In contrast with its origin in recreational mathematics, graph theory today uses sophisticated combinatorial, probabilistic and spectral methods with deep connections with a variety of areas in mathematics and computer science. In this chapter appendix, some vibrant new directions in graph theory have been selected and described to illustrate the richness of the mathematics involved as well as the utilization through major threads of current technology. The list of the sampled topics is by no means complete since these areas of graph theory are still rapidly developing. Abundant opportunities in research, theoretical and applied, remain to be explored.

References

D. Achlioptas, R. M. D'Souza and J. Spencer, Explosive percolation in random networks, Science, 323, no. 5920, (2009), 1453–1455.

W. Aiello, F. Chung and L. Lu, A random graph model for massive graphs, Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, ACM Press, New York, 2000, 171–180.

M. Ajtai, J. Komlós and E. Szemerédi, Largest random component of a k-cube, Combinatorica 2 (1982), 1–7.

D. Aldous, The random walk construction of uniform spanning trees, SIAM J. Discrete Math. 3 (1990), 450–465.

N. Alon, I. Benjamini, and A. Stacey, Percolation on finite graphs and isoperimetric inequalities, Annals of Probability, 32, no. 3 (2004), 1727–1745.

R. Andersen, F. Chung and K. Lang, Detecting sharp drops in PageRank and a simplified local partitioning algorithm, Theory and Applications of Models of Computation, Proceedings of TAMC 2007, 1–12.

R. Andersen, F. Chung and K. Lang, Local graph partitioning using pagerank vectors, Proceedings of the 47th Annual IEEE Symposium on Founation of Computer Science (FOCS'2006), 475–486.

J. Balogh, B. Bollobás and R. Morris, Bootstrap percolation in three dimensions, Annals of Probability, to appear.

J. Balogh, B. Bollobás and R. Morris, The sharp threshhold for r-neighbour bootstrap percolation, preprint.

Y. Bao, G. Feng, T.-Y. Liu, Z.-M. Ma, and Y. Wang, Describe importance of websites in probabilistic view, Internet Math., to appear.

A.-L. Barabási and R. Albert, Emergence of scaling in random networks, Science 286 (1999), 509–512.

P. Berkhin, Bookmark-coloring approach to personalized pagerank computing, Internet Math., to appear.

E. K. Blum and S. V. Lototsky, World Scientific Pub. Co, 2006, pages 333–337.

B. Bollobás, C. Borgs, J. Chayes, O. Riordan, Percolation on dense graph sequences, preprint.

B. Bollobás, Y. Kohayakawa and T. •Luczak, The evolution of random subgraphs of the cube, Random Structures and Algorithms 3(1), 55–90, (1992)

B. Bollabás and O. Riordan, Robustness and vulnerability of scale-free random graphs. Internet Math. 1 (2003) no. 1, 1–35.

C. Borgs, J. Chayes, R. van der Hofstad, G. Slade and J. Spencer, Random Subgraphs of Finite Graphs: I. The Scaling Window under the Triangle Condition, Random Structures & Algorithms, 27, (2005), 137–184.

C. Borgs, J. Chayes, R. van der Hofstad, G. Slade and J. Spencer, Random subgraphs of finite graphs: II. The lace expansion and the triangle condition, Annals of Probability, 33, (2005), 1886–1944.

C. Borgs, J. Chayes, R. van der Hofstad, G. Slade and J. Spencer, Random subgraphs of finite graphs: III. The phase transition for the n-cube, Combinatorica, 26, (2006), 395–410.

S. Brin and L. Page, The anatomy of a large-scale hypertextual Web search engine, Computer Networks and ISDN Systems, 30 (1–7), (1998), 107–117.

K. Chaudhuri, F. Chung and M. S. Jamall, A network game, Proceedings of WINE 2008, Lecture Notes in Computer Science, Volume 5385 (2008), 522–530.

J. Cheeger, A lower bound for the smallest eigenvalue of the Laplacian, Problems in Analysis (R. C. Gunning, ed.), Princeton Univ. Press (1970), 195–199.

X. Chen and X. Deng, Settling the complexity of 2-player Nash-equilibrium, The 47th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2006, 261–272.

F. Chung, The heat kernel as the pagerank of a graph, Proc. Nat. Acad. Sciences, 105 (50), (2007), 19735–19740.

Fan Chung at the AMS-MAA-SIAM Annual Meeting, January 2009, Washington D. C.

F. Chung, A local graph partitioning algorithm using heat kernel pagerank, WAW 2009, Lecture Notes in Computer Science, vol. 5427, Springer, (2009), 62–75.

Fan Chung, Graph theory in the information age, Notices of AMS, 57, no. 6, July 2010, 726–732.

F. Chung, P. Horn and L. Lu, Diameter of random spanning trees in a given graph, preprint.

F. Chung and P. Horn, The spectral gap of a random subgraph of a graph, Special issue of WAW2006, Internet Math,.2, (2007), 225–244.

F. Chung, P. Horn and L. Lu, The giant component in a random subgraph of a given graph, Proceedings of WAW2009, Lecture Notes in Computer Science, vol. 5427, Springer, (2009), 38–49.

F. Chung and L. Lu, Connected components in random graphs with given expected degree sequences, Annals of Combinatorics 6 (2002), 125–145.

F. Chung and L. Lu, The average distances in random graphs with given expected degrees, Proceedings of National Academy of Sciences 99 (2002), 15879–15882.

F. Chung, L. Lu and V. Vu, Eigenvalues of random power law graphs, Annals of Combinatorics 7 (2003), 21–33.

F. Chung, L. Lu and V. Vu, The spectra of random graphs with given expected degrees, Proceedings of National Academy of Sciences 100 no. 11 (2003), 6313–6318.

F. Chung, L. Lu, G. Dewey and D. J. Galas, Duplication models for biological networks, J. Computational Biology 10 no. 5 (2003), 677–687.

F. Chung and L. Lu, Coupling online and offline analyses for random power law graphs, Internet Math. 1 (2004), 409–461.

F. Chung and L. Lu, Complex Graphs and Networks, CBMS Lecture Series, No. 107, AMS Publications, 2006, vii + 264 pp.

F. Chung and L. Lu, The volume of the giant component of a random graph with given expected degrees, SIAM J. Discrete Math., 20 (2006), 395–411.

F. Chung and S.-T. Yau, Coverings, heat kernels and spanning trees, Electronic Journal of Combinatorics 6 (1999), #R12.

Douglas E. Comer,"Internetworking with TCP/IP, Volume 1: Principles, Protocols, and Architecture," Prentice Hall, 1995Also see G. Held, Ethernet Networks , 1994, John Wiley for earlier ethernet-based versions.

C. Daskalakis, P. Goldberg and C. Papadimitriou, Computing a Nash equilibrium is PPAD-complete, to appear in SIAM J. on Computing.

P. Erd˝os and A. Rényi, On random graphs, I. Publ. Math. Debrecen 6 (1959), 290–297.

P. Erd˝os and A. Rényi, On the evolution of random graphs, Magyar Tud. Akad. Mat. Kutató Int. K¨ozl. 5 (1960), 17–61.

M. Faloutsos, P. Faloutsos, and C. Faloutsos, On power-law relationships of the Internet topology, Proceedings of the ACM SIGCOM Conference, ACM Press, New York, 1999, 251–262.

R. Feldmann, M. Gairing, T. Lucking, B. Monien and M. Rode, Selfish routing in non-cooperative networks: A survey, Lecture Notes in Computer Science, vol. 2746, (2003), pp. 21–45.

A. Frieze, M. Krivelevich, R. Martin, The emergence of a giant component of pseudo-random graphs, Random Structures and Algorithms 24, (2004), 42–50.

M. R. Garey and D. S. Johnson, Computers and Intractability, A Guide to the Theory of NP-Completeness, W. H. Freeman and Co., San Francisco, 1979, x + 338 pp.

E. N. Gilbert, Random graphs, Annals of Mathematical Statistics 30 (1959), 1141–1144.

G. Grimmett, Percolation, Springer, New York, 1989.

S. Hoory, N. Linial and A. Wigderson, Expander graphs and their applications, Bulletin of AMS, 43, no. 4, October 2006, 439—561.

J. Hopcroft and D. Sheldon, Manipulation-resistant reputations using hitting time, WAW 2007 , LNCS, vol. 4863, (2007), 68–81.

G. Jeh and J. Widom, Scaling personalized web search, Proceedings of the 12th World Wide Web Conference WWW, (2003), 271–279.

M. Jerrum and A. Sinclair, Approximating the permanent. *SIAM J. Comput. 18*, (1989) 1149–1178.

M. Kearns, S. Suri and N. Montfort, An experimental study of the coloring problem on human subject networks, Science, 322 no. 5788, (2006), 824–827.

H. Kesten, Percolation theory for mathematicians, volume 2 of Progress in Probability and Statistics. Birkh¨auser Boston, Mass., (1982).

J. S. Kong, N. Sarshar and V. P. Roychowdhury, Experience versus talent shapes the structure of the Web, PNAS, 105, no. 37 (2008), 13724–13729.

S. Kontogiannis and P. Spirakis, Atomic selfish routing in networks: a survey, Lecture Notes in Computer Science, vol. 3828, (2005), 989–1002.

David Liben-Nowell and Jon Kleinberg, Tracing information flow on a global scale using Internet, PNAS, 105, no. 12, (2008), 4633–4638.

L. Lovász and M. Simonovits, Random walks in a convex body and an improved volume algorithm, Random Structures and Algorithms 4 (1993), 359–412.

L. Lovász, Random walks on Graphs, Combinatorics, Paul Erdős is Eighty, Vol. 2, Bolyai Society Mathematical Studies, 2, Keszthely (Hungary), 1993, 1–46.

C. Malon and I. Pak, Percolation on finite Cayley graphs, Lecture Notes in Comput. Sci. 2483, Springer, Berlin, (2002), 91–104.

M. Mihail, Conductance and convergence of Markov chains-a combinatorial treatment of expanders, Foundation of Computer Science, (1989), 526–531.

M. Mitzenmacher, A brief history of generative models for power law and lognormal distributions, Internet Math. 1 (2004), 226–251.

M. Molloy and B. Reed, A critical point for random graphs with a given degree sequence. Random Structures & Algorithms 6 (1995), 161–179.

M. Molloy and B. Reed, The size of the giant component of a random graph with a given degree sequence, Combin. Probab. Comput. 7 (1998), 295–305.

N. Nisan, T. Roughgraden, E. Tardos and V. V. Vazirani, Algorithmic Game Theory, Cambridge University Press, 2007.

A. Rényi and G. Szekeres, On the height of trees, J. Austral. Math. Soc., 7 (1967), pp. 497–507.

T. Roughgarden, Selfish Routing and the Price of Anarchy, MIT Press, 2006.

T. Roughgarden and E. Tardos, How bad is selfish routing? Journal of the ACM, 49, (2002), 236–259.

Walter Savitch, "JAVA", second edition, Prentice Hall, 2001

D. Spielman and S.-H. Teng, Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems, Proceedings of the 36th Annual ACM Symposium on Theory of Computing, (2004), 81–90.

Richard Stevens, TCP/IP Illustrated, Volume 1: The Protocols," Addison Wesley, 1994

References for Computer Networks: Sockets and MPI

Ran Giladi, Network Processors, Morgan Kaufmann, 2008

W. Gropp, E.Lusk, A Skjellum, Using MPI, 1994, MIT Press, Cambridge, MA

G. Held, Ethernet Networks, 1994, J. Wiley, N.Y.

Marc Snir et al, MPI The Complete Reference, 1996, MIT Press, Cambridge, MA

W.R. Stevens, UNIX Network Programming, 1990, Prentice Hall

B. Quinn and D. Shute, Windows Sockets and Network Programming. 1996, Addison Wesley.

G. Wright and W.R. Stevens, TCP/IP Illustrated, vol.2, 1995, Addison Wesley.

Video-streaming

O.K. Tonguz and G. Ferrari Ad Hoc Wireless Networks: A Communication-Theoretic Perspective, John Wiley, May 2006.

X. Zhu and B. Girod, Distributed rate allocation for multi-stream video transmission over ad hoc networks, Image Processing, IEEE International Conference on ICIP 05, vol.2, 2005, pp 157–160.

Chapter 8
High Performance Computing and Communication (HPCC)

James M. Pepin

Introduction

As announced in the hardware Chap. 5, there is a relatively new paradigm of computation in which a single very large application problem is partitioned into subproblems which can be computed concurrently by being distributed over a network of computers, called a *cluster*, usually a large-scale local area network (LAN) in which *nodes* of the network can be connected directly through large switches to build a *message-passing system*. The given application problem is computationally intensive and would take too long to solve on an existing single computer, however fast it may run. For certain problem structures it is possible to partition the problem into components which can be run concurrently to achieve a speed-up that makes the solution feasible. An example is the solution of a linear system Ax = b where A is a nxn matrix and b is an n-dimensional vector. The Jacobi iterative method can be programmed to work on K groups of m components of x concurrently where n = Km. The K groups are computed on K nodes of the cluster. The individual K computers installed as the nodes in the network, usually pc's, operate independently on their assigned subproblems for the most part, but at various points in their computations they must communicate with each other by a message-passing protocol which permits the cooperative exchange of intermediate results needed in the various subproblems. This paradigm is designated as high performance computing and communication (HPCC).

This chapter describes both the technical and managerial problems in HPCC. One of the technical problems is how to program an application in a format suitable for concurrent computation. In this chapter we give only one example of how this can be done by a programmer making use of MPICH, a library of subroutines that provides sophisticated procedures for message-passing between the nodes computing the

J.M. Pepin (✉)
Clemson University, Clemson, United States
e-mail: pepin@clemson.edu

E.K. Blum and A.V. Aho (eds.), *Computer Science: The Hardware, Software and Heart of It*, DOI 10.1007/978-1-4614-1168-0_8,

subproblems. In Chap. 9, some general programming techniques are given for distributed/parallel execution of large applications.

The class of HPCC computing paradigms described in this chapter are characterized as being implemented by installing complex hardware switches connecting a large local area network of pc's and a related message-passing software library such that these systems do not require or use the Internet protocols described in Chap. 7 on Computer Networks.

HPCC at the University of Southern California designates both the paradigm and the physical facility where it is practiced, the center for High Performance Computing and Communication. We acknowledge the assistance of Maureen Dougherty in providing data on the USC HPCC facility implemented on the campus at the University of Southern California (USC). This facility exemplifies the many versions of HPCC implemented at major universities and research labs.

At USC, the HPCC center was founded in 2000 as a resource dedicated to supporting intensive research computing and the networking capabilities needed to achieve it. HPCC was and is part of the campus information technology (IT) organization. Before HPCC was created the campus IT functions were supported as part of the academic computing mission of the campus IT organization. At USC campus IT merged with the engineering school IT support group in the mid-1980s. The combined function delivered IBM mainframe computing, departmental and research group focused mini-computers and mini-supercomputing resources. As the computing and networking landscape increased in complexity and scope USC determined that a tightly focused HPCC center would provide a competitive advantage to researchers engaged in large-scale computing.

USC had provided large-scale research computing with three large mini-super computer systems (HP-SPP, IBM-SP2 and SGI-2000) at the end of the 1990s. At that time there was a shift in the focus of large-scale computing away from purpose-built super systems toward collections of commodity-based hardware connected in local networks called *clusters*. Also USC was the ‘regional network’ service provider for higher-education sites in the region via the Los Nettos research and education network. These services were different enough from ‘production IT’ services like mail, web and other day-to-day functions that the transition to a stand-alone cluster was a logical step.

All of these services and components make up what the National Science Foundation (NSF) is calling Cyber-Infrastructure (CI). CI is becoming a critical piece of the national science infrastructure. Over the past 10 years computing, data management and communications have become a larger part of the toolkit that scholars are using to move science forward. The most recent report from NSF is called CF21, Cyber-infrastructure Framework for the twentyfirst century. It describes a CI eco-system which spans lab and departmental resources, campus-wide resources such as HPCC and national resources like the Teragrid HPCC environment. The emergence of “team science” projects like the Large Hadron Collider and other national and international collaborations requires a new approach to computation which leads many campuses to develop organizational structures like HPCC.

HPCC at USC, as mentioned earlier, evolved as a unit of the campus IT structure. At some universities the evolution of research support for large-scale computation may have not been part of the campus IT organization. Each university has a different organizational culture. There are three common themes that have evolved supporting CI. The first is the one used at USC, described below. The second is one based on supporting research CI from an organization base in the Vice Provost (or Vice President) office for research. This model allows focusing on research needs on a campus where the IT group is built as a production-based IT 'plumbing' support group. Many of these campuses do not have a Chief Information Officer (CIO) who has leverage across the entire campus community. The third support model is one that totally distributes the research support back to the individual research groups. This is commonly seen at very large research universities with 'star' researcher(s). These three templates show up in institutions focused on research, commonly known as the Research1s (R1s). The rest of the academic community (comprehensive colleges, 2 year schools etc.) is diverse in their CI support. In many cases they have minimal central funding of IT itself and much of the leading edge work will depend on a single faculty member or researcher to provide leadership and support.

The USC HPCC Center

The USC HPCC Center environment includes two computer clusters, disk storage facilities, an Ethernet network, large-scale tape sub-systems for data backup, Myricom switch high performance interconnect networks, software support and computer room power and A/C infrastructure.

The Clusters

The two computer clusters are two generations of the same management model. The individual computers (pc's) in a cluster are called *nodes*. The first cluster was built starting in 2001 and consisted of xeon and AMD nodes that were 'two- flop/ clock' computers, the standard 'pc' chip of the time. It has 1,044 nodes. The "two-flop" designation means that a node is able to process two floating-point instructions per process clock cycle. The second cluster employs four-flop/ clock nodes (the current standard). When the flops/clock changed this started the build of the second cluster. If one mixes nodes of different rates of processor flops in the same cluster, then one has problems with developing software that runs in multiple cores/node with different processing modalities. The second cluster has 1,732 nodes and an increased number of cores per processor as well as flops/clock. The recent evolution of commodity-based chips has been toward an increase in the number of cores per processor and flops/clock while not increasing the frequency of the chip. This avoids the increased heat that would need to be dissipated by chips as the frequency is increased. The number of transistors will increase, allowing more cores and more parallel execution but the raw execution speed of a chip will not increase. This fundamental shift from the past evolution where clock speeds were routinely increased has created the cluster systems of today as the 'standard' way of doing HPCC research computing. The use of 'commodity parts', e.g. using PC chips, because they are low cost has caused disruption in the monolithic advances in performance based on faster single processors. It has also created a 'programming' or 'algorithm' crisis. Legacy codes and techniques, in many cases, do not map directly onto systems with 1,000 s of loosely connected heterogeneous pc cores.

The advent of heterogeneous 'pile of nodes' clusters has created requirements for many new and innovative management and design techniques. In the past, cluster system designers and support staff would only have to procure, configure and support one (or a few) systems. This meant there was only one operating system instance to support and update. Today a large cluster can have thousands of nodes where each node may be running its own operating system on a unique piece of hardware. All of these nodes have to be managed; i.e. inter-connected, deployed among users and debugged. USC uses Xcat to manage the nodes. Xcat is an open-source systems software package developed by IBM. It allows the cluster system manager to deploy operating systems in an automated way, and configure network addressing and other routine tasks that would usually be done in a single operating system environment and distribute them to the nodes. USC has also deployed console services on all the nodes so that the system administrator can see and act on routine messages about nodes. Xcat also can be used to start and stop the entire cluster or portions of the cluster. To effectively use the cluster a *node-access scheduling* system is deployed. At USC this is the system called PBS/Maui/Torque. Analogous to an OS scheduler (Chap. 6), this software suite works at the node level and takes user requests for cluster services and schedules access to nodes to *user jobs* based on number of nodes requested, memory usage, time required and other usage parameters. There are many different access-scheduling systems for clusters. PBS is the one USC selected in the early days and maintains since changing scheduling systems would require significant operational changes for the user community.

Cluster Network Switch Fabrics

The USC HPCC clusters have network switch fabrics to support message-passing between nodes and access to file systems. An Ethernet network is used to connect nodes to file servers. This uses the Ethernet network interface cards and cable connections (NICS etc.) that are standard on a commodity PC (See Chap. 7). The early nodes had 100 Mb/s Ethernet circuits on the system cards but more recently this has been upgraded to 1 Gb/s Ethernet (See Ethernet discussion in the **Computer Networks** Chap. 7).

As stated at the outset, a cluster provides a paradigm and platform for large-scale computing problems which run too slowly on a single computer for practical results. The platform is usually a network of commodity computer nodes such as PC's. The paradigm requires the user to decompose a large problem into smaller subproblems which can run independently on the nodes except at points at which they must exchange intermediate data results before they can continue. The data exchange is done by *message-passing* (MP) according to a standard known as MPI (Message-Passing Interface). (See the Computer Networks Chap. 7). For internode message-passing, the clusters at USC provide network connections by a *switch fabric* which employs Myrinet switches connecting Myrinet *links* (full-duplex Myrinet cables). (See the *Guide to Myrinet-2000 Switches and Switch Networks, Myricom Inc., revision 27 August 2001.*) Switch fabrics are an art as well as a technology and Myrinet switches enhance the art by permitting a variety of fabric topologies. The first cluster was based on Myrinet switch fabrics providing 2 Gb/s connections and the newer cluster is based on 10 Gb/s connections. Myrinet *switch fabrics* are usually *Clos-based networks*, a multi-stage network topology (introduced by Charles Clos, Bell System Technical Journal, March 1953) described in an example below. An advantage of a Clos network such as Myrinet's, is that it reduces packet latency compared to Ethernet, at the cost of a more complex switch fabric. Large-scale benchmark cluster programs using Clos networks in Myrinet-2000 switch enclosures can be 30–40% more efficient in execution of message-passing than ordinary switching networks. As explained in the cited Myricom Guide, a Myrinet-2000 switch is built as a package (or *enclosure*) that has as basic building blocks a 16-port *crossbar* switch, which is implemented on a single chip, the XBar16. A *crossbar* switch is perhaps the simplest switch in switching circuit design, consisting of so-called horizontal wires (or *bars*) conducting what can be thought of as **input** signals and such horizontal lines being *crossed* by vertical lines (or *crossbars*) for conducting **output** signals. The XBar16 chip has 16 horizontal bars $i = 1 - 16$ for inputs and 16 vertical bars for output lines, $j = 1 - 16$. At each possible junction (i, j) of a horizontal bar i and a vertical crossbar j, there is a switch which can connect line i to line j, say a transistor switch (see Logic Circuits, Appendix to Chap. 5) which is normally "open" (non-conducting). To connect input line i to output line j the transistor switch at (i, j) is closed by applying a control signal which makes the transistor conduct. In some Myrinet switch package networks, only some of the 16 horizontal and/or vertical bars in an XBar16 are used, as in the package described below. It should be recognized that in a crossbar switch (or any other) the roles of "input" and "output" lines in a circuit are reversible in that once a switch at (i, j) has been closed circuit signals can flow from vertical "output" line j to horizontal "input" line i.

As an example of the application of a Myrinet *enclosure*, consider a cluster of only 128 nodes for which it is required to be able to connect any cluster node to any other. The following Myrinet multi-stage Clos network switch enclosure can be used. (We give a verbal description since a diagram is rather intricate and not easy to follow. In this case the proverbial picture may not be not worth a 1,000 words. However, the reader is encouraged to draw a connection diagram according to the following verbal instructions.) As an initial Clos network *stage*, there are 16 XBar16 chips installed on 16 circuit *cards,* say cards i = 1,. . .,16 with eight plugin *ports* on each card providing a total of 8 × 16 = 128 ports for connecting cables making connections to the 128 cluster nodes. These cards are inserted into support slots in a "lower stage" of the enclosure box. The "back" of the enclosure houses another eight XBar16 chips. These back crossbar chips are arranged in a *backplane spine* "upper stage". Eight vertical bars of the chip on each of the 16 initial stage cards are connected so as to *fan-out* in the ratio 1:8 so as to cross each of the eight spine chips horizontal bars. (Draw a fan-shaped set of eight lines emanating up from each initial card.) As seen from each upper stage card there is a total of 16 different lines from the 16 initial cards *fanning in* to each upper stage spine chip. Finally, the 16 vertical bars of each of the spine chips are connected **back down** in a 1:16 fan-out shape as input lines on each of the 16 initial cards. The resulting switch fabric is called a *spreader* network in the Myrinet Guide, since (if you draw the diagram of connecting lines) the lines look like two sets of spreading fans, one set of fan lines going up from each of the 16 cards to the eight spine chips and the other set of fan lines coming down from each of the spine chips to the 16 cards. This spreader network of lines provides a conducting path from any node port to any other node port when the appropriate switches are closed. There is obviously a unique shortest switch path between two nodes connected to ports on the same card. There are also eight possible *minimal paths* between any two nodes A and B connected to ports on different cards i and k, respectively say. These minimal routes traverse three XBar16 switches: first from the node A port through a switch on card i up to some available (unused) spine chip, j say, and second through a spine switch on chip j back down to the designated card k and third through the switch on k to the node B port. There are eight possible minimal switch paths (one for each value of j) between any two nodes A and B, so that the message handling capacity (i.e. concurrent message-passing paths possible between all pairs of nodes) is as large as possible for the total of 24 XBar16 switches in this Clos enclosure. Large-scale benchmark cluster jobs using such Clos networks can be 30–40% more efficient in available concurrent message-passing between nodes. Since most user applications use sub-clusters smaller than the total cluster size this is acceptable. There are several other interconnection technologies but USC has used Myricom switching for a long time and has direct relationships with the Myricom founders, one of whom was from USC-ISI. This has also allowed USC to directly collaborate with the developers of the MPICH system (message-passing middle-ware; see below).

HPCC Disk Storage

On-line disk based storage is one of the most important components of a cluster and one of the hardest to design and deploy. The range of I/O requirements in a modern cluster environment creates severe design tradeoffs. Many applications are legacy codes from desktop environments that are being scaled up to clusters.

This causes application to I/O mismatch. If a job requires ten small files on a small desktop but is going to run on 1,000 nodes, it will require 10 files × 1,000, that is 10,000 files active for that single job. Modern file systems (see Chap. 10) are not efficient at handling millions of small files in a directory structure. Network based file systems like NFS are not good at scaling to thousands of simultaneous accesses. One can deploy local file access on each node but then moving data to and from the active nodes is not simple. The first cluster used simple NFS based file access with SAM/QFS file systems. USC also deployed PVFS (a parallel file system) and experimented with many variations. There is an ongoing research effort in the national centers that has created file systems like Lustre, but each of the efforts has similar tradeoffs around small files with large directories. There is also an interesting trend that is doubling disk space every year or 2. This trend has compounded the problem. Backup of data has become a significant headache as well. USC uses on-line tape libraries with tapes that can hold 1 TB (terabyte) of data each but the upward trend in disk has created a capacity race between offline media (tape) and disk. The cost curve still does not make disk 'cheaper' and spinning disk costs power. Another issue with disk storage is the same that has been seen in processors. Modern disk technology is driven by commodity usage. This means disks are designed around desktop or laptop environments where failures are, to some extent, tolerable. When disks were mainframe quality the failure rates were commensurate with the use. Today there are 100 s or 1000 s of disks in a disk pool and the failure rates on SATA disks are high. This means designing RAID environments that take this into account, with the corresponding complexity and maintenance requirements.

Heat and Air Conditioning

The current generations of clusters are creating a serious problem in data centers due to the 'heat density' of the clusters. A rack of 40 nodes (what normally fits in a data center rack based on two socket servers) will consume 15 KW of power at peak load. This is at the edge of what traditional air-cooling, using raised floor techniques, can achieve. Before USC HPCC moved to a new data center there were serious problems with random hot spots because of low airflow. The new data center that HPCC occupied in early 2007 employed air flow from the ceiling (60%) and below floor A/C (40%). This was enabled by the high ceiling in the renovated facility and a 24 in. high raised floor. Many data centers do not have this ability. The new data center was built with 1.2 Mw of power potentially in two 5,000 sq ft dedicated HPCC areas. Again this is a fairly unique environment. Many research facilities are in the 1Mw range for all of their uses based on data centers built 20–30 years ago. The increase in density of servers is also creating hardship on many campuses that do not have central facilities to house their clusters. If one puts 100–200 Kw systems in traditional academic buildings the power and air conditioning systems are not able to handle this without major upgrades. Departmental systems are causing increased costs for building modification as well as running A/C systems 24/7. To reduce the airflow requirements new rack cooling techniques are becoming popular. IBM and other vendors are delivering water-cooled-door based systems, including containment isles and other techniques to reduce the requirement for larger Computer Room Air Conditioning (CRAC) units. These doors circulate chilled water though coils that the air leaving the racks pass through. This design can make a rack up to 30 KW of heat dissipation heat neutral. Since the cooling is done at the rack the power savings achieved by not having to circulate large volumes of cold air from CRAC units is significant.

Financing Clusters

USC pioneered a cluster business model called 'condo clusters'. The business model actually started at USC in the late 1980s with 'minicomputers' that a department or research group would purchase and the university would match resources to run the systems in the campus computer center. The advent of clusters made this model even more logical to employ. Due to the significant facilities costs to operate and house a cluster in an academic building the central IT group was able to justify paying for the basic infrastructure of the cluster. This includes network high-speed interconnection, racks and support costs like power, A/C and systems support. The condo 'owners' buy the compute nodes or storage and install them in the facilities which central IT procures. The HPCC acts as the purchasing agent twice a year and the participating research groups buy nodes at this time. The campus IT also purchases some nodes for use by non-sponsored groups and students. The cluster owners and general community participate in a shared governance environment via a HPCC advisory group, plus allocated university procured node usage via an allocation committee. The central IT budget also covers professional management of the clusters. This is a significant advantage. Research groups do not have to sacrifice the careers of a graduate student or post-doc to be a systems administrator.

OS and Applications Interface Software

A modern computational resource, as noted in Chap. 6, must have an Operating System (OS). It also needs certain applications interface software. The USC cluster nodes use Linux as the OS (see Chap. 6) and have a full suite of applications interface software. One of the key pieces of software is MPICH, which is an implementation by Argonne National Labs of the Message Passing Interface (MPI) library. MPI provides a set of interfaces that are used to transmit data and synchronize data transmission between processes on nodes in the cluster. Simple examples include broadcasting and point-to-point communication. (There is a MPICH2 version that is in development as well.) MPI enables concurrent running of thousands of cooperating processing *threads* of an application program across the nodes of the cluster. It is aware of the capabilities of the high performance network switching fabric and takes advantage of any ability to reduce message communication latency and provide high performance data sharing between the distributed elements of the application program. The USC cluster accommodates a wide range of programming languages including C and Fortran. Applications software linking BLAS, Matlab and other subroutines are installed. If a cluster user requires special software packages, the systems staff will install and support packages that groups require. Some of the applications are developed to work in a distributed concurrent cluster mode with thousands of nodes but some still are optimized for supercomputers of years ago with vector hardware. This is a serious programming problem that the community is grappling with today.

HTC and HPC

There are two basic classes of computing that are done on the clusters; High Throughput Computing (HTC) and High Performance Computing (HPC). HTC jobs are typically one thread (core) or at most several per execution. They can however use massive amounts of resources as they run for long periods of time or are run many times with different parameters. HTC jobs do not require high performance interconnects for message passing but could require lots of I/O. HTC jobs also sometime create file system bottlenecks due to a large number of files being created or processed at the same time. HPC jobs take advantage of the parallel nature of the cluster. They may use a large number of cores (threads) to execute a single job. HPC jobs require (in many cases) high performance interconnect. To become HPC versus HTC new algorithms will be needed. This will speed up the results from tasks that are run on single cores today. In the past this was done by clock rate increase. Now it is done by parallelization of the computation.

The simple example of a numerical application suitable for HPC is the classical Jacobi iterative method for solving a large linear system $Ax = b$, where A is an nxn matrix with large n. If $x^{(m)}$ is an iterate which is an approximation to x, then the next Jacobi iterate is given in matrix–vector formulation by

$$x^{(m+1)} = D^{-1}(D - A)x^{(m)} + D^{-1}b, \text{ where D is the diagonal of A.}$$

(e.g. see Numerical Analysis and Computation, by E.K. Blum, Addison-Wesley 1972 or Dover 2011). In a component formulation it is easily seen that each ith component $x_i^{(m+1),}$, $i = 1, \ldots,$ n involves only the previously computed components of iterate $x_{.}^{(m)}$ Hence all new ith components of $x^{(m+1)}$ can be computed concurrently. For large n, say $n = Km$, the iteration computation can be broken down into K sections each dealing with m components and the sections allocated to K pc's in a cluster. Each pc updates its designated m components concurrently with the other pc's. When its update computation is completed, each pc will send its new updated components to all other pc's for the next iteration. This is done by calling the appropriate message-passing subroutines in the MPICH library.

To address the changes in programming models required by large-scale clusters, the academic community needs to develop programs to train the next generation of scholars in the computational techniques that run in massively parallel cluster environments. This means developing inter-disciplinary teams from mathematics, computer science and the various disciplines using the resources. As we move to Petascale (1,000 trillions operations per second) and then Exascale (1,000× increase again) the current programming models may not scale to the millions of cores that will populate a Peta or Exa scale system. The distribution of data between diverse memory islands and the latency created by this distribution mean that traditional applications developed for vector-based supercomputers 20 years ago will break down. Those models were based on close affinity of memory to the processing elements and data streams that were near clock speeds. New systems will also be based using accelerators such as General Purpose Graphics Processing Units (GPGPUs). The GPGPUs will be popular for the same reasons commodity PC processors have dominated. The commodity eco-system for games and portable devices require this kind of product.

Clouds

We have focused our attention on HPCC implemented by clusters organized as LANs with sophisticated switching fabrics. We have not considered possible Internet versions of HPCC. *Cloud computing* (see Appendix to this chapter) is an evolving HPCC model using resources that are dispersed (like clouds) over many large computing facilities and possibly accessed remotely, plus leveraging on a massive scale by using various management techniques such as *virtualization*. The USC HPCC center can be viewed as a form of a cloud limited to a single facility that provides *Infrastructure as a Service*. (IaaS). (See Appendix.) There are various definitions of a cloud, one of which requires an on-demand virtualization presentation to the end users. Aside from the obvious networking problems discussed in the chapter on Computer Networking, there are other serious technical issues in cloud computing, such as the speed of light not being negotiable, that must be taken into account in the design and use of a cloud model. Clouds can be regarded as an evolution from the Grid model of computing that started in the late 1990s and timesharing systems that existing since the 1960s. The grid model was combining resources at disparate locations with scheduling software to move batches of work around the various sites. Timesharing was the gold standard of interactive use in the 1970s. There was a rich landscape of very sophisticated systems. They included operating systems like Multics, Tenex, Tops10, Tops20, VAX/VMS and MVS/TSO to name a few. With the advent of the IBM 370 architecture virtual systems also become practical. In the 1980s it was common to run multiple operating systems on one large mainframe with VM/370 and Logical Partitions (LPARs). The hardware on the mainframe was optimized to support the virtual instances, something that today's microprocessors are just achieving. Cloud computing resources, especially when they cross international borders, create security and legal control questions. Who controls legal access, what are the restrictions on use and other similar vexing questions are being worked out today. The international governance aspect of the Internet and cloud computing are a fertile field for lawyers and politicians to harvest. Today the security requirements for applications like medical records create conflict with the cloud paradigm. Transnational privacy differences also cause conflicts. Some countries require access to all data for security reasons and others prohibit exactly the same requirements.

Data storage technologies are also a driving factor in cloud computing. The cloud provides both advantages and disadvantages. Using large distributed data centers it is easy to replicate data across diverse geographic locations. Using file system techniques a user will not have to explicitly worry about disaster recovery at the physical layer. However, the cloud provider does have to be aware of dependencies they create. A file system must be careful not to mirror inconsistent data across sites; using traditional mirroring techniques are not effective across long distances; that pesky speed of light thing creates file latency inconsistencies. The cloud operators must also design file systems that allow for massive numbers of files. This is difficult and research needs to be done. The problem is the same that HPCC centers are seeing when a user creates massive numbers of small files. Perhaps the most important part of these massive file systems is the chance of data corruption due to transmission of data across computer networks and use of massive numbers of commodity disk drives. New versions of RAID are required and TCP/IP as a transport protocol is known to be vulnerable to bit errors when terabytes of data are moved and the underlying network has potential undetected bit errors.

Finally to use a cloud environment data needs to be moved from the source (campus) to the cloud centers. This will require new and improved optical wave based paths. These paths are over distance (speed of light problem again) and have significant cost when it leaves academic network environments. If one looks at the power of computation available on a typical research campus 40 years ago and compares that to the bandwidth of the connection to the campus then extrapolates to today, the storage and computing capacity has gone up 10^7 and the network connection bandwidth 10^5. Going forward, storage and processing power is doubling every 18 months to 2 years while network bandwidth is doubling maybe once every 10 years. There is a crisis in broadband capabilities in the United States that we are not addressing. This only addresses the external network issues. The explosive growth in capabilities of instruments to record data and of cheap disk to store it has also created a massive local data problem. We have genetic sequencing equipment appearing around a campus that can create TBs of data in hours and we don't have a set of procedures in place to store, organize and protect the data. This is just one example of new equipment enabled by advances in technology that are out-stripping our ability to rationally manage.

The previous paragraphs may give one pause about Cloud Computing, but there are many positive aspects. A cloud environment is very similar to the aggregation of resources that HPCC provides inside USC. The use of professionally managed resources in large-scale data centers is very compelling. These data centers can be sited in locations where low cost and perhaps green energy is available. USC in Los Angeles pays roughly 10 cents per KW/h of power. At Clemson University in rural South Carolina power is 4.5 cents per KW/h. These differences can create a compelling cost model for cloud computing or aggregated HPC. Power densities of large-scale clusters are only increasing while the cost of the hardware is decreasing (per computing unit). The consumer of cloud or remotely provisioned HPCC needs to be aware of the complexities it can create.

Some Conclusions

The HPCC systems discussed above provide a useful model to describe the relationship between departmental, campus, regional and national CI. The evolution of networks, and software and hardware capabilities have created a complex set of interrelated systems. The CI eco-system has become the workspace for day-to-day use of computing by researchers and scholars. There are systems that range from wireless devices to High Performance Computing centers. To bridge between these systems a campus must pay attention to evolving standards in networking, computing, data-storage and identity management. To not participate in the national and international HPCC environments would mean a campus is at a competitive disadvantage. Centers like HPCC at USC provide the middle-level glue that plugs the gap between a research group and resources they need. HPCC helps users with usage of large campus based resources, while providing a coherent path to national and international resources when the campus level resources are not adequate. This includes job scheduling, applications development and network provisioning. Having a group that participates in the national scene means researchers can focus on their work and allow HPCC staff to be their extension into the national centers.

HPCC centers can be an enabler for creation of Virtual Organizations (VOs). VOs are a cyber instance of an affinity group. Examples of VOs abound in the commercial sector (facebook, myspace etc.). In the academic space we have many communities who have created environments in a similar vein. The Southern California Earthquake Center (SCEC) is an example of a VO combined with a physical organization. There are dozens of others around the world. The VO revolution is a logical extension of the academic societies and organizations that have existed for centuries. The VO can create instant feedback and community building. Along with VOs, 'portals' are being developed to allow disciplinary access to complex HPCC based resources. A scholar will connect to a web portal or use special laptop or remote device software to interact with complex software systems. This mode of access allows the scholar to use software models that are prepackaged to create results based on their input.

HPCC also enables leading edge network services and research at the university. USC was one of the earliest participants in the ARPAnet in the early 1970s. Participation in advanced networking research, deployment and development has been ongoing since that time. USC was one of the first universities to deploy campus wide networks and deployed one of the first regional networks, Los Nettos, in the country via USC-ISI and campus IT staff. Los Nettos is still an active research and education network, providing services for USC, Caltech, JPL and other academic sites in Southern California. The ability to innovate new networking capabilities is a mandatory complement to the computational and storage resources. Scholarship and especially research has become a team sport. It is more common for a physics researcher at USC to interact with some physicist in Europe than with a chemist in the next building. These relationships are enabled by the next generation techniques such as wave division multiplex enabled connections to national networks. The same drivers to innovate are present in the networking space as computation. The requirement to reach out and embrace disruptive change in networking is hard for a campus production IT organization. An organization like HPCC provides the impetus for new solutions that a production shop would shy away from. HPCC has been a leader in new services in the region and has partnered on many international efforts via a joint project called Pacwave. Pacwave is a distributed fiber based network consortium that reaches from Seattle to San Diego, and enables specially provisioned wavelengths between researchers on the west coast to the rest of the country and on to the rest of the world.

Appendix

Cloud Computing

E.K. Blum

In the preceding chapter a new computing paradigm, called *cluster computing* was presented. It involves distributing a computational problem over a computer network. The chapter also referred to a paradigm called *cloud computing*, which has been promoted recently. Cluster computing under certain conditions of management can be regarded as an example of cloud computing, since both paradigms involve distributing computation over an assemblage of computational resources. However, the term *cloud computing* is not well-defined if at all. The purpose of this brief appendix is to clarify the current meaning of the term.

A description has been formulated by the National Institute of Standards (NIST) and we are inclined to agree with the NIST description coauthored by Peter Mell and Tim Grance. We give a brief summary of their description, with our own evaluations added. We note that there have been several articles on Cloud Computing published in the ACM Communications, for example in 04/2010, vol.53, No.4, pp.50–58. However, these articles fail to give a precise definition and are, like the following account, only a sketchy outline. This reflects the newness and rapidly changing formations of clouds.

We agree with NIST that cloud computing is an "evolving paradigm" and we further understand that it is at present (and possibly for the immediate future) incompletely defined. At present, the cloud computing industry (those organizations which claim to be engaged in cloud computing) works with many different models and approaches. Indeed some vendors seem to confuse it with cluster computing, which is discussed in our preceding chapter on high performance computing (HPCC). Indeed, cluster computing may be regarded as a case of cloud computing under certain conditions. This indicates that the "cloud" aspect, like clusters, certainly involves a distribution of computing resources. How wide or organized this distribution can be remains unspecified. Most models of cloud computing assume a cloud service provider which provides a shared pool (a "cloud") of computing resources (servers, storage units, networked pc's, application software etc.) in a manner that can be rapidly accessed and released by a client with minimal overall management or service provider action. An authorized client of a cloud service can unilaterally request and acquire resources as needed, such as server time, storage space etc. without approval of the request by the cloud service provider. Presumably the quantity of resources available in the cloud permits this. The access to networked components of the cloud is through standard networking techniques that allow mixed client platforms of devices like mobile phones, laptops, pc's and PDA's. The cloud service provider can serve multiple clients by providing resources in a dynamic mode according to changing clients demands. Note that this dynamic allocation of resources to clients is also a feature of large clusters. (See the preceding chapter on HPCC.) The quantity of resources is usually so large as to seem virtually unlimited to the client. Resource utilization can be monitored and reported to the service and the client. NIST recognizes three explicit categories of cloud service as follows.

Cloud Software as a Service (SaaS). This service allows a client to use applications software already running on the cloud infrastructure. Various client devices (e.g. pc's or laptops) can access applications software as web pages through a web browser. *Cloud Infrastructure as a Service (IaaS*). Infrastructure such as storage networks or computer networks is provided to a client on which the client can run arbitrary software of his own, including operating systems and applications. The client does not control the infrastructure except for possible limited control of some network components such as host firewalls. *Cloud Platform as a Service (PaaS*). Client has control over deployed applications but cannot manage/control cloud infrastructure. There are four models of cloud deployment: (1) *Private*: infrastructure dedicated to a singe organization; (2) *Community*: infrastructure shared by several cooperating organizations; (3) *Public*: infrastructure available to the general public and owned by a company selling cloud services; (4) *Hybrid*: a mix of two or more of the first three but using a proprietary technology that permits load balancing between the three. Despite its lack of precise definition cloud computing is an active new mode of computation. For further examples see Chap. 9.

Chapter 9
Programming for Distributed Computing: From Physical to Logical Networks

Christian Scheideler and Kalman Graffi

The first programming languages predate computers and were mostly used as a form of mathematical reasoning. With the advent of modern electronic computers, programming languages became prominent as tools to specify and control the behaviour of these machines. A programming language is an artificial language that makes use of the functions that can be performed by a machine and allows one to express algorithms precisely. Early languages (see Chap. 4) were Fortran (1957) and Algol (1958), which were used for numerical computations. Cobol was issued in 1962 and optimized for business applications. Many other influential languages emerged from the late 1960s to the late 1970s, among them Simula (which introduced object-oriented programming), C, Smalltalk, Prolog (the first logic programming language) and ML (realizing a polymorphic type system on top of Lisp). Each of these languages spawned an entire family of descendants, and most modern languages count at least one of them in their ancestry.

The capabilities and characteristics of a programming language define the type of computer processes that are contemplated. The von Neumann computer architecture (Chap. 5) established a basic model for sequential computation processes. It postulates (Chap. 4) a computer organization based on a single separate central processing unit (CPU) and a separate memory unit accessible to the CPU. Having single or multiple CPUs or memory units introduces a classification of various types of computers and related programming approaches. Due to trends in the design of computer chips, nowadays, multiple CPUs and multi-threading (sequences of computation steps) on one chip are de-facto-standards in computers. This trend emerged in 1963 when the first time-sharing systems appeared.

Multi-processor machines introduced various challenges for the concurrent access of programs to the shared CPUs and memory. With the advent of the Internet era and the trend towards cluster computing (Chap. 8), the distribution of computing devices

C. Scheideler (✉) • K. Graffi
Department of Computer Science, University of Paderborn, Paderborn, Germany
e-mail: scheideler@upb.de; graffi@mail.upb.de

E.K. Blum and A.V. Aho (eds.), *Computer Science: The Hardware, Software and Heart of It*, DOI 10.1007/978-1-4614-1168-0_9,

and the memory widened even further. A cluster of computers offers many distributed computational and memory resources, more than an individual computer. Operating systems were organized to provide services during the execution of a program, such as links to various built-in system subroutines. This is called the *runtime env*ironment. With a suitable computer language and runtime environment, distributed resources can be harnessed and conveniently used on single applications. For more details, see Chap. 8 on high performance computing.

In this chapter, we elaborate on general principles of distributed and parallel programming and discuss specific solutions for distributed computing in physical networks as well as in application-layer networks, which are also known as *overlay* networks.

Distributed and Parallel Programming

A *distributed* or *parallel* computer system consists of a family or set of autonomous computers organized to interact so as to cooperatively execute a computation, such as a cluster (Chap. 8). The autonomous computers offer either single or multiple computational resources. In the case of each autonomous computer having also a private memory, we speak of a *distributed* system. In this case, the computers interact through a network in which they exchange messages and synchronize their actions. (See the HPCC Chap. 8). If the autonomous computers have access to a *shared memory,* we use the term *parallel* computing. In this case, communication is performed via the shared memory.

The motivation for the use of parallel or distributed systems is two-fold. First, an application may impose distributed characteristics by creating and consuming data in differing physical locations. Such a characteristic is, for example, found in the various email applications or in typical (client-server) consumer-provider based applications. The second reason for distributed/parallel computation is motivated by the practical benefits in comparison to the use of a single computer. In comparison to a single computer, a cluster of computers may be extended in number and thus extended in the resources available. There is no intrinsic scalability limit in the clustering of the computers. *Server farms* of up to 100,000 computers exist. Through scalable and redundant (software and hardware) architectures, the damage effects of a single point of failure can be eliminated. However, in order to realize the full potential of a distributed/parallel system, a programming language is needed that has primitives for specifying the effective use of the available resources.

Levels of Parallelism

As discussed in Chaps. 3 and 4, a *program* is a list of instructions to be executed on a computer aiming at performing a specific task. In its simplest execution mode, its instructions are run sequentially on a CPU, one after another, storing intermediate and final results in a single memory. However, to use the power of a large number of computers, the program must be split up into smaller parts that are run *concurrently,* or as we say, in parallel. Various levels of parallelism exist that define the *granularity* by which a program may be split up. The highest level of granularity is on the *program* level. On this level, various programs may be run independently on multiple computers, but no tasks within a program may be shifted or run on other computers. The next level of granularity supports parallelism in the execution of *procedures*. In this case, instruction sets, encapsulated in procedures, may be transferred (or *called)* to ex*ecute* in parallel at remote computers. At this level, coarse parts of programs may be placed so as to execute on more suitable computers. At the *instruction* level, individual operations (e.g. arithmetic operations) may be run in parallel to more efficiently use the available resources. At a lowest level, the *bit* level, the internal execution of a single instruction is performed in parallel, e.g. on a multi-core CPU or a GPU with many *stream* processing units. With this grain of parallelism, individual atomic operations can be speeded up.

While very fine-grained levels of parallelism in computer programs are best performed in hardware, the parallelism of instructions and procedures is best exploited in software by distributing the operational load of the execution of a program over various *cores* or computers. To obtain full access to the distributed resources, the procedures or instructions may be either dispatched or called manually by the programmer or may be run automatically by the runtime system environment. Both of these approaches to parallelize applications have been heavily investigated in the literature. The automated approach requires recompiling programs with special parallel compilers in order to enable their parallel execution. This is convenient for the programmers as existing programs do not need to be adapted and no further work is needed. However, automated parallelization often lacks efficiency, as it is very challenging for a compiler to optimally split the program into parallelizable tasks. Using the second approach, the programmer needs to learn how to deal with parallel programming patterns and explicitly apply the parallelization instructions to exploit parallelism. While this approach is more time-consuming in the creation of the programs, it usually results in a much more efficient parallel execution of the code.

Tasks in Parallel Programming

For both approaches, the implicit and explicit parallelization paradigms, a set of management tasks needs to be executed by the runtime environment in order to enable the parallel execution of a program. According to Kasim et al. (2008) the set of available computers (*workers*) needs to be managed, the *workload* needs to be partitioned into tasks, the *tasks* mapped to workers (computers) and the intermediary results need to be synchronized. The first step comprises the management of the workers. It is needed to supervise and monitor the resource utilization of the workers as well as the connectivity and contact details of the workers in the case where they are distributed. On the other hand, the program must be split into sections of concurrently runnable *processes* and prepared for parallel execution. This is a non-trivial programming design problem as the semantics of the program may change subtly if certain processes are executed prior to others. This step is either done manually by the programmer or automatically by the runtime environment. Given the workers and the individual program *processes*, a *mapping* is needed. The mapping from processes to workers must take the abilities of the workers into account as well as the requirements of the processes. In the case of shared memory, the tasks in the memory are accessible from all CPUs. In the case of a distributed system, the processes are either pre-deployed and remotely accessed or need to be distributed in advance to the corresponding workers. Finally, the results of the calculations of the processes need to be synchronized and joined. For distributed computing all the processes need to be suitably addressed. In the following, we present how these tasks are addressed in physical networks. Specifically, we present concepts for parallelism on computers and networks, give examples and introduce selected programming languages that ease the distributed programming. (See Chap. 8 for related discussions.)

Physical Computers and Networks

In this chapter, we distinguish between distributed systems based on actual physical networks and virtual systems based on logical networks. In the first case, computers or *cores* are locally close to each other and connected directly. This case also comprises the hardware-based multi-core computers, in which the CPUs are also closely linked. In this case, the CPUs and memory are close to each other, so the main questions are related to how the access to the resources is managed. We discuss the main questions related to distributed systems based on logical networks in the next section.

In order to utilize the resources several programming approaches are in use in practice. One main classification criterion is the involvement of the programmer. On the one hand, the programmer may explicitly decide which instructions should be run in parallel. On the other hand, this decision may be handled implicitly under the assumption that the entire code is parallelizable. Given that the assumption is true, the runtime environment may optimize the code execution on its own using various programming concepts outlined below.

Concepts of Parallel Programming

Next, we describe a selected set of concepts for *parallel programming*. In the early days of parallel programming, single CPU devices supported subroutines called *co-routines*. In order to use these, the programmer had to actively trigger a co-routine operation. Co-routines are subroutines that can be run in parallel and may be paused. However, this approach is limited to single-core devices and cannot be extended to multi-core computers.

The *fork and join* approach is another of the earliest parallel programming concepts. The names are suggested by flowcharts. The *fork* operation in a program marks the beginning of a parallel process, while the *join* operation synchronizes the results. Applying the fork operation in a UNIX OS environment copies the process that is forked and enables each process to identify its status through checking its process identifier. Forked processes are run in parallel and combine their results in a matching blocking *join* call.

The *remote procedure call* (RPC) is an approach to integrate remotely located procedures. Procedure calls for remote and local code have the same syntax. RPCs are resolved transparently for the programmer. The runtime environment of the programming language offers client and server *stubs* as local code. These stubs are in charge of managing the network connections, marshalling and de-marshalling the messages and managing blocking operations.

While the previous concepts require the programmer to use parallelism explicitly, *implicit parallelism* denotes a different parallelism concept. This concept assumes that every *process* can be run in parallel except for parts called *critical codes*. Thus, the execution of a process may be interrupted and run on a different core or computer. In order to protect *critical code* parts from being disrupted, the programmer may set critical code in a *synchronized blo*ck of code.

Examples of API's for Parallel Programs

Most of these concepts have been implemented, deployed or discussed in the literature. In the following, we introduce successful APIs that gained a lot of attention in recent decades.

The software tool *parallel virtual machine* (PVM) is an API designed to allow a (heterogeneous) network of computers to be used as a single distributed parallel computer platform (as elaborated in Sunderam (1990)). Using PVM, program *tasks* or *independent threads* of instructions may be run in parallel on the virtual machine using explicit remote invocation. Through explicit message passing based on task identifiers, the tasks may communicate with each other. The PVM author group offers libraries of such parallelizing devices for the C, C++ and Fortran languages and wide support for a large set of devices.

The *Message Passing Interface* (MPI) is explained in Pacheco (1996) and is a de-facto standard for developing high performance computing applications on parallel and distributed memory architectures. (See Chap. 8.) It specifies message passing (MP) operations for *point-to-point* message passing between pairs of processors and also for *collective* message-passing between groups of processors. It provides libraries of MP subroutines for C, C++ and Fortran as well. MPI requires the programmer to mark individual processes (tasks) in the program and map the processes to workers (computers). The user of the program may define how many processes are assigned for the execution of a program. For common modes of communication between processes, MPI supports point-to-point and collective communication based on message passing. In addition, a barrier analogy is used to synchronize the processes. The barrier operation requires all specified processes to pass the barrier and thus to finish their tasks before the execution of the processes can continue. Thus, programming entails some intricate planning of the paths of concurrent execution.

Several theoretical models for distributed programming concepts have been presented in the literature. Lipton and Sandberg introduced an abstract machine termed *Parallel Random Access Machine* (PRAM) in Lipton and Sandberg (1988) which models the various types of concurrent read and write operations in a shared memory system. It abstracts from synchronization and communication issues, but gives a tool at hand to investigate the effects of various distributed programs.

Bulk Synchronous Programming (BSP) was introduced by Valiant as a model taking the synchronization and communication overhead into account. BSP introduces three stages to model parallel computing. In the first stage, a set of processors with local memory is assumed. These processors perform local operations and do not interact. In the second stage, communication between the processors takes place and results are exchanged. Finally, in the third stage the processors are barrier synchronized by waiting until all processors finish a particular communication. As the two last stages are expensive, the idea is to perform as much local computation as possible in the first stage so that the overhead generated by the other two stages is insignificant. Obviously, too much overhead would incur delays that could cancel any speed-up expected from concurrent execution. This is true of any parallelization technique. The stages are continuously repeated until the program terminates. This model has been intensively discussed and extended in the literature, and several implementations exist, (see Bonorden et al. 2003). A few examples are the logP, QSM and HMM models.

Another formalization of interactions in a concurrent system is given by Hoare, and termed *Communicating Sequential Processes* (CSP), (see Hoare 1978). Such a formalization helped to specify and verify the correctness of various concurrent programs.

MapReduce was introduced by Google in Dean and Ghemawat (2008) to facilitate the processing of large data sets on a set of distributed computers. Libraries of the software framework have been developed for various programming languages. In the *map* step, a problem is split into smaller sub-problems and assigned to a set of workers, which themselves may split the assigned problems further, creating a subproblem tree structure of the problem. Each worker reports its results to its father, which combines in the *reduce* step the individual results of its children workers to its own result. With this approach, data-intensive and loosely coupled computational problems can be solved very efficiently by a large set of workers. This scenario is often the case in cloud environments like those in Google's or Amazon's server farms.

While these examples represent certain standards or models, another approach to parallel programming is defined by the set of available concurrent (and parallel) programming languages. With a specific parallel/concurrent programming language at hand, a programmer can exploit parallelism more efficiently. Several such programming languages have been introduced in the last few decades. We present some of the most influential ones below.

Concurrent Programming Languages

In the early days of software engineering, the discipline of programming was mainly concerned about the increasing complexity of the programs. Early structured programming languages like *Pascal*, *Fortran* and *Cobol* introduced *structured programming* as a method to create reusable, dependable code. However, they were limited to numerical problem solving in their practical applicability. Starting with 1962, the idea of information hiding and abstract data types shifted the focus from numerical programs to object-oriented programming, which was first used in the *Simula* language.

Concurrent programming languages were first discussed with the new time-sharing machines and later since 1975 with the advent of the personal computers and their organization into networks. Some of the object-oriented programming languages were extended to support the new trend of parallel programming, such as *Smalltalk* (1972) or *Ada* (1973). Previous programming languages did not sufficiently support the parallel execution of programs. *Occam* was introduced in 1983, following the strict formalization of Hoare's CSP. *Erlang* was introduced by Ericsson in 1986, supporting concurrent programming using an *actor model*, which relies on message passing between processes rather than shared variables. However, with the Internet era and the World Wide Web, *Java* started its success story in 1996, when it was introduced and supported as the programming language for the WWW. (See Chap. 4.) Due to its application field and usability, Java had a large impact both in industry and academia. Several extensions for secure distributed programming emerged, leading to new programming languages such as *E*. The language *E* originates from Java and uses message-passing, event loops and promises for managing the concurrency in the programs. Another programming language that relates to Java is *Clojure*, which runs on the Java Virtual Machine. However, *Clojure* is a functional programming language, combining the benefits of *LISP* and *Java*. One of the youngest programming languages is *Go* developed and announced in 2009 by Google. *Go* is similar to *C* but with structures for concurrent programming. *Go* is inspired by Hoare's CSP. However, unlike *Occam*, *Go* does not support verifiable or safe concurrency.

In the history of concurrent programming languages, specific patterns recur. Message passing is a de-facto standard, as the alternative shared memory, mutexes or semaphores are technically infeasible or inefficient. As an example, we introduce the programming language *Erlang* to show the main principles for concurrent computing based on message passing. (See also Chap. 8.)

A Brief Introduction to Erlang

Erlang was developed around 1986 by the Ericsson company as a programming language that supports concurrent programming, focusing on highly available programs in the area of telecommunications. Its main approach to increased availability is to strictly isolate the individual processes in a program, avoiding any means of shared memory. The processes are, however, allowed to communicate using message passing, both with processes on local cores but also with processes on remote machines. An essential requirement for safe message passing is the usage of *pure protocols*, which do not use any pointers or information which might be bound to a specific machine. Fault tolerance is further reached by allowing machines and processes to crash. Instead of trying to catch any possible exception in a process and to react to it internally, an invalid process is left to crash. However, the crash of any process is communicated and detected by a monitoring process, which initiates actions upon the observation of such a failure. For that, the monitoring machine needs sufficient information to decide on the appropriate next steps. In particular, it needs information on the cause of the crash. Thus, fault tolerance is reached with replication and a safe-to-crash approach. This approach was considered inefficient back in the 1980s, especially due to the limited bandwidth and duplication of program state information (instead of using a shared memory). Nowadays, the main bottleneck in the execution of a program lies in missed cache hits in the case that illegal pointers are used. The approach of duplicating the program information for message passing fits the current trends of computer system architectures.

An *Erlang* program typically consists of thousands of processes, divided into *worker* and *monitoring* processes. Worker processes have links to monitoring processes, which form a error-propagation channel in the case that the worker process fails. This classification of process types is described in the *Erlang* Open Telecom Platform system, which suggests monitoring trees that hold a protocol that is initiated once a failure is detected. A failure of a working process is interpreted as the inability to perform a desired computation. Thus, a new (easier) approach for solving the problem is typically instantiated.

Erlang offers in its first implementation a *Prolog*-like programming syntax with *variables, atoms, tuples* and *lists*. In order to support concurrent programming, message passing is modelled by a "P ! M" command, meaning that the message *M* is sent to the process with the process id *P*. Knowing the process ids is a must in *Erlang*. The receive operation *receive* is called in a non-blocking way at the addressed process. It matches the received message with a list of patterns and initiates the corresponding action upon a match. In order to fork a process to run in parallel to the calling process, the *spawn* command is to be used. It initiates a new process and reports its process id, which can then be used as a communication address.

Error detection of remote or local processes is set up using a method called *link (PID)*. This call links the current process to monitoring the behaviour of the process with process id PID. In order to receive these out of band messages, the monitoring process is flagged as a system process using the command *process_flag(trap_exit, true)*. Upon an irregular termination of the monitored process, an error message is created containing the keyword *Exit*, the process id of the broken process and a reason, ('Exit', PID, Reason). This message is passed to the monitoring processes and evaluated there. This simple approach allows to create large and complex applications. Message passing is used both between large components, but also within a large component between the smaller components it consists of. Any component of an application can be tested whether it produces expected results. Components producing unexpected or false results can be opened and tested recursively until the root of the failure is found.

Erlang has been in use at Ericsson to operate ATM telecommunication switches in a reliable and fault-tolerant manner. With the Open Telecom Platform, a set of libraries has been created that provides all essentials of an application middleware package, easing the use of Erlang for large and complex programs. Parallel to the advent of Erlang, further trends of parallel and distributed computation in the Internet have been observed. These trends were not influenced by programming languages, but more by novel distributed architectures and principles. *Logical networks* were formed, granting direct access to distributed resources, allowing new approaches to parallel and distributed programming.

Logical Networks

For a long time, hardware (physical) networks, realized either on a chip or among a set of hardware devices, were used for distributed programs. The main assumption was that all of these devices were controlled by the same person or institute. (See Chap. 7 on clusters.) Later, in the Internet (Chap. 7), a client-server (software) architecture was put in use to support such popular applications of WWW as FTP and email. This is a logical network overlaying the physical one. The main idea was that many users access a centrally hosted server and exchange their information over this central mediator. Local code and functional roles were more and more shifted to servers in the Internet, leading to weak and simple clients on the user side. This logic network approach allows the operation of specialized machines for certain tasks to buy and sell computation services in the Internet and to increase both efficiency and productivity.

The concept of *service oriented architectures* (SOA) was introduced in the late 1990s as an approach to organise software architectures i.e., logic networks. Its main idea is to map business processes to *workflows*, which are decomposable into smaller service units. Individual services may then be either programmed by the user himself or bought in a (not yet existing) global marketplace for services. Through the clear separation and specification of single services, a marketplace may be born, leading to professional and well tested service components as well as flexibility in the design of system architectures. A further extension of this idea came about recently with the advent of c*loud computing*. (See Chap. 8, Appendix.)

While SOA is limited to the hosting and execution of remote code, cloud computing offers a wider set of services. As already remarked in Chap. 8, Appendix, cloud computing can offer software as a service (SaaS), platform as a service (PaaS) and infrastructure as a service (IaaS). We elaborate below.

At the end of this chapter, we summarize the research and industrial efforts in the area of *peer-to-peer* (p2p) networking. p2p systems are distributed architectures created through establishing a logical network between peers typically connected to each other via the Internet. The peers are mostly autonomous computer installations under control of individual users. Thus, it is challenging to provide the functionality of a desired application with a desired quality. This is a developing field of distributed programming.

Service-Oriented Architectures: A Survey

Service oriented architectures (SOA) extend the idea of open interfaces and RPCs in such a way that the components or services to be used are offered in a world-wide marketplace. Thus, productive and well-programmed services may be rented and used to compose complex software architectures. SOA promises to make the creation of software architectures more flexible and thus allow a fast adaptation of these architectures to business needs. Business workflows are a main driver in SOAs. These workflows define the steps in business procedures controlled by computers and are partitioned into individual services, which are composed from the set of available services in the service market place. Individual services may be exchanged for cheaper or more productive service implementations. In order to dynamically do this, a set of automated actions need to be performed. A service provider needs to *publish or register* his service implementation at a service directory like UDDI (Unified Description, Discovery and Integration). The service consumer (client) needs to *find* a service (in the directory) that matches its needs. Match making is a challenging task even though service descriptions use standardized web service description languages (WSDL). These are not always explicit on defining the semantics of a service. Once a suitable service is found, it is *bound* to the service consumer. This is done by the service directory delivering a contact address for that service to the service consumer. As a final step, the service is *executed* in a typical RPC manner by sending the input values in a well-formatted manner, typically using the protocols SOAP (Simple Object Access Protocol) or REST (Representational State Transfer).

While SOA does not introduce a new programming language for parallel or distributed computing, it parallelizes the programming and service consumption itself. Programs or program components do not have to be written by the same person or loaded manually. Services are searched and bound automatically to the runtime file of the program, picked and paid at an internal or global market place. The service composition can be optimized according to various characteristics, ranging from the costs to the service level agreement details. The quality of service provided by every service is specified, as well as the API and costs for using it. While a SOA may be easy to implement on the scale of an organization, a global market place is still to come. Many users and enterprises hesitate to release their services to avoid showing internals of their business workflows. Also, enterprises fear to use the services of other providers in order not to reveal internal workflow information by transferring revealing input values in a service call. An extension of the SOA idea from software services to platform and infrastructure services has recently had a large impact under the term "the Cloud". See Chap. 8 Appendix.

Cloud Computing-A Survey and Critique

Cloud computing (Chap. 8 Appendix) emerged consequent to the trends of SOA, as a method of on-demand and pay-per-use services with regard to a wide set of resources. While SOA focuses only on software as a service, with the advent of *grid computing* (large networks of existing distributed computer resources) and *utility computing* platforms for application hosting or infrastructure resources themselves can be reserved and used nowadays over the Internet. A great driver for these trends lies in the *virtualization* of hardware devices, allowing systems to host or port several instances of software programs. Thus, with the demand and access patterns for a specific software or hardware service, the corresponding hosting resources can now be increased or decreased on the fly, without the interruption of the running program.

The emergence of Web 2.0, an extension of the classical World Wide Web that allows interaction with websites (Chap. 7) has accelerated the shift of personal user data to the Internet, where it could be modified or hosted using various novel Web 2.0 applications. *Software as a service* (SaaS) as one aspect of the cloud provides applications in the Internet, both to be used by users directly (e.g. Google Docs) or over specified APIs (e.g. Open Social). Having the user data in the Internet (e.g. Facebook) allows the direct manipulation and the central hosting of applications for it (so-called *social networking*).

A further aspect of the cloud comprises the offering of platforms, or *platform as a service* (PaaS). In this case, vendors offer a rich set of functionality to support the life cycle of individual services to be hosted in the Internet. The life cycle support ranges from application design and development, testing, deployment and hosting, as well as background services like storage, monitoring and accounting. Platforms offer Web service designers the freedom to offer their services and pay only for the consumed resources, for which they also can charge their customers. This process liberates the designers from predicting and reserving hardware resources that their application might use in peak times. The elasticity of the cloud, i.e. adapting the resource provision to the demanded level, is a key benefit for both the application designers, operators and users. However, several limitations on program structures were defined by the vendors. Computation State is considered harmful, as it cannot be elastically ported on the virtualized computers.

From a commercial point of view, most cloud providers offer very competitive prices in comparison to self-operated server farms. This comes at the danger of lock-in and being bound to a specific cloud vendor. It still remains challenging and work demanding to change the programs hosted on one cloud to be runnable on another cloud. Due to differing commercial interests, cloud vendors aim at hindering the support of interoperability.

A third layer in cloud computing offers more freedom to its users. *Infrastructure as a service* (Iaas) offers core computational and storage services on demand, leaving the usage and management of these resources up to the service consumer. Examples of the various layers of cloud computing are the Google App engine, Amazon's Elastic Compute Cloud (EC2) and Microsoft's Windows Azure Platform.

p2p Networks-a Survey and Critique

While a small set of cloud players were trying to build centralized server farms to offer any kind of services a user needs, an orthogonal trend emerged with the advent of Napster, BitTorrent and Skype. This trend of peer-to-peer (p2p) networks provides services and applications through the creation of logical networks and distributed mechanisms mainly or solely using the resources of the consumer devices. In contrast to the previously mentioned approaches, p2p systems consist mainly of user devices that are autonomous and out of reach of a provider. Thus, the resources are freely available, as are the users, but are unreliable and of fluctuating quality. In addition, due to the large number of autonomous nodes, networks operating on p2p resources must be *self-organising* and consider the fluctuation of node resources and presence. This paradigm has had a large impact in the area of file sharing applications. In 1998 *Napster* offered a centralized index which acted as a publish/subscribe platform to share and search for information files among the users. The subsequent file transfer after a positive match was performed directly from peer to peer. Due to the vulnerability of the centralized index, subsequent file sharing applications used a logical network, an overlay, to interconnect the peers and to implement the search for desired files or objects.

A set of overlays has been proposed both in academia and open source communities. This set can be classified into structured and unstructured overlays. Structured overlays, like *Chord*, *CAN* or *Pastry*, place the objects stored in the overlay by a method based on their object IDs according to a predefined structure. Thus, the insertion of objects requires some time, but the retrievability of these objects is guaranteed and the lookup time is low. In an unstructured overlay, like *Gnutella 0.4* or *GIA*, objects remain at the peers, thus produce no additional overhead for insertion, but the search for these objects is rather costly. In the worst case, all peers in the network have to be contacted to find a desired object.

A further classification aspect of overlays is whether all the peers have the same roles or whether there are some peers with special roles and more responsibilities. Super-nodes are used typically in hybrid overlays like *Gnutella 0.6* to maintain the state of normal nodes. Most overlays are flat, meaning that all peers are organized in only one overlay. Hierarchical overlays which consist of various protocols for various layers of the network, e.g. like in *Omicron*, are rare. Most popular overlays nowadays implement a *distributed hash table* (DHT), which was first introduced in the Chord paper by Stoica et al. (2001). The main idea is to split up a hash table and distribute it over all peers. Maintaining the links to the neighbouring parts of the hash table as well as shortcuts in the hash table to peers in exponentially increasing distances allows traversing the hash table consistently and quickly to find the desired entry. Dabek et al. (2003) a common API for structured p2p overlays which extends the common lookup functionality of hash tables. Several overlays follow that principle, like *Chord*, *CAN*, *Pastry* or *Tapestry*. With the distributed nature of the overlay, the most critical challenge to be addressed is the quality fluctuation of the peers, which requires mechanisms to repair the structure of the overlay all the time.

p2p networks offer access to the wide set of resources that comes with the user devices. In contrast to centrally hosted applications, even the cloud, which reaches its limits with the increase of users, p2p systems benefit from an increase of the number of users, as the pool of resources increases as well. p2p networks are mainly used for file sharing applications, in Skype for free Internet telephony or in some applications, like *PPLive*, for live and on-demand video streaming. However, a broad service offer like in the cloud is not yet feasible in p2p networks due to the limited control of the quality of these networks. The quality fluctuates with the quality of the resources offered by the users. This is a characteristic that is unsuitable for most productive applications. Although there is a vast amount of resources to use and to harness, research has not yet devised the tools and mechanisms to contend with the quality fluctuation and to provide reliable quality based on the user resources contributed to the p2p networks.

Conclusions

Distributed programming emerged with the advent of multicore and networked computers and reaches out to the trend of globally distributed software development. Nowadays, with more and more computational resources distributed on a single machine, at home locations, in widespread enterprises or even over the world, tools and paradigms are needed to handle the desired programs and problems in a way that harnesses the potential of these distributed resources. Distributed programming languages give a tool at hand to create suitable programs that are parallelizable, distributed runnable and still fault-tolerant and reliable. In this chapter we described the characteristics and classifications of distributed programming and briefly introduced the features of the programming language Erlang. In the future, however, system architecture will play a larger role than it did in previous decades. With globalization, software projects are globalizing. Software components are envisioned to be bought and bound on a global market place, corresponding operational platform resources are rented on demand to run those services and distributed programming emerges to become the art of distributed service composition. On a small scale, however, the craftsmanship and art of distributed programming needs to be further investigated and improved in order to create the building blocks of society's key tool for business prosperity: the IT industry.

Reference

O. Bonorden, B. Juurlink, I. von Otte, and I. Rieping. The Paderborn University BSP (PUB) library. *Parallel Computing*, 29:187–207, February 2003

F. Dabek, B. Y. Zhao, P. Druschel, J. Kubiatowicz, I. Stoica: Towards a Common API for Structured Peer-to-Peer Overlays. IPTPS 2003:33–44

J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, 2008

C. A. R. Hoare: Communicating Sequential Processes. Commun. ACM (CACM) 21(8):666–677 (1978)

H. Kasim, V. March, R. Zhang, and S. See. Survey on Parallel Programming Model. In, *Network and Parallel Computing*, volume 5245 of *Lecture Notes in Computer Science*, pages 266–275, 2008

R. J. Lipton and J. S. Sandberg. PRAM: A Scalable Shared Memory. Technical Report, TR-180-88, Princeton University, Dept. of Computer Science, Sept. 1988

P. S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996

I. Stoica, R. Morris, D. R. Karger, M. F. Kaashoek, H. Balakrishnan: Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. SIGCOMM 2001:149–160.

V. S. Sunderam. PVM: A Framework for Parallel Distributed Computing. *Concurrency – Practice and Experience*, 2(4):315–339, 1990

R. L. Wexelblat (Ed). History of Programming Languages. Academic Press, 1981

Chapter 10
Databases

Michael Benedikt and Pierre Senellart

Introduction: Two Views of Database Research

This chapter is about *database research* (or as we abbreviate *DBR*). To people outside of computer science – and perhaps to many within – it will be unclear what this term means. First of all, what is a "database"? Used generally, it could mean any collection of information. It is obvious that there are deep scientific issues involved in managing information. But information and data are very general notions. Doesn't much of computing deal with manipulating data or information? Isn't everything data? Clearly the databases that DBR deals with must be something more specific.

Database research takes as its subject something more specialized: *database management software*, a term which we will use interchangeably with *database management systems* (DBMSs or "database managers"). This refers to a class of software that has emerged to assist in *large-scale manipulation of information in a domain-independent way*. By *domain-independent*, we mean to contrast it with, say, a software package that calculates your taxes, or tools that help show your family tree – these are definitely processing data, but are manipulating it in ways that are very specific to one dataset (set of data). In contrast, there are many complex tasks that are associated with *storage*, *update*, and *querying* in a "generic sense". As early as 1962 software products emerged that were dedicated to providing support for these tasks. A particular concern was with performing them on large amounts of data. A current DBMS can filter (read and select) gigabytes of data in seconds, and can manage terabytes (10^{12} bytes) of data. In the last few decades a vibrant industry has sprung up around DMBS tools and tool suites. For example, personal computer users would know the DBMS Microsoft Access;

M. Benedikt (✉)
University of Oxford, Oxford, UK
e-mail: michael.benedikt@cs.ox.ac.uk

P. Senellart
Télécom ParisTech, Paris, France
e-mail: pierre.senellart@telecom-paristech.fr

E.K. Blum and A.V. Aho (eds.), *Computer Science: The Hardware, Software and Heart of It*, DOI 10.1007/978-1-4614-1168-0_10,

software developers would know the major commercial vendors, such as Oracle and IBM, along with open-source database management systems such as MySQL.

The relationship of generic database management products to end-to-end applications has varied over the years. Many companies advertise software that provides simply "generic database management". In other cases DBMS software is bundled either in application suites, or with e-commerce suites that also handle issues that have little to do with data. Similarly at runtime the relationship of a DBMS to the rest of the application could take many forms; there could be a dedicated database management process communicating with other software processes via a well-defined protocol. Alternatively, DBMS functionality could be available as libraries. Regardless of these "packaging" issues, database management software can be considered a separate "functional entity" within an application. A systems programmer requires special training to create it, wherever it sits. Someone (an application programmer or an end-user) must interface with it, and often in doing so must understand how it is engineered and how to tune it.

So there is a special kind of software called *database management software*. In a narrow sense, DBR refers to *the scientific study of this software*. But why should there be a field of database research? You may grant that database software is important. But there is no theory of tractors or knapsacks, despite their utility and fairly well-defined scope. In the software domain there is no field of "spreadsheet theory", or "word processor research", at least not one of comparable significance. Why does a DBMS warrant a separate research area?

We give two answers to this question. First of all, the design of a DBMS is complex and the approaches to creating a DBMS are stable enough and specific enough to the setting to be amenable to scientific study. Software of major database vendors runs to hundreds of thousands of lines of source code, and has evolved over a period of decades. Spreadsheets are also complex, but the tools and design principles used are either not stable enough, or not unique enough to the spreadsheet setting to sustain a separate discipline. Thus we can refine the definition of DBR as the study of the stable architectural and component construction of database management systems – the fundamental languages, algorithms and data structures used in these systems. *Database theory*, a subset of DBR, would then be the formalization and analysis of these languages and algorithms, e.g., semantics of the languages, upper and lower bounds on the time or space used in the algorithms. The emphasis on *stable* components explains why many features in a DBMS are not the subject of much research – there are comparatively few research papers about report generators, administrative interfaces, or format conversion for their lack of stability.

The complexity and uniqueness of software dedicated to data management gives one justification for DBR. However, a deeper motivation is that database management techniques and algorithms have become pervasive within computer science. Many of the features of modern database systems that we shall discuss in this chapter – *indexing*, *cost-based optimization*, *transaction management* – that were first developed in the context of database management systems have become essential components in related areas as well. The study of logic-based languages, which received its impetus from the success of relational databases, has had impact on understanding the relationship between declarative specification and computation throughout computer science. Thus much of DBR is concerned with the application and expansion of “large-scale data management techniques” wherever they are or could be relevant in computer science. This “migration” accounts for a fact which will be fairly obvious to the reader of the proceedings of database conferences: *much of current DBR is not closely connected to current DBMS product lines at all.* Much of DBR involves languages that do not exist within current DBMSs, or features that go radically beyond what current systems offer. They may deal with proposed extensions of real languages, or modeling languages that are put forth as theoretical tools but which are unrealistic for practical use.

This phenomenon is not unique to DBR. A significant portion of programming language research investigates ideas for novel languages or language features, and security research often looks at the possible ways in which security might be achieved, regardless of their practicality. Similarly, DBR examines the ways in which computers *could* manage large quantities of information, rather than how they do manage it. Thus, much of DBR deals with “managing information” in a very wide sense. It concerns itself with broad questions: How can new information sets be defined from old ones? How could one describe relationships between datasets, and how could we specify the information that a user or program might want from a collection of structured information? What kinds of structure can one find in information? What does it mean for two sets of information to be the same? In DBR, these questions are dealt with from a computer science perspective: precise description languages are sought and algorithmic problems related to these description formalisms are investigated.

We see that DBR has a manifold structure; much of it revolves around existing database management systems, while another aspect revolves around techniques for data management in the wider sense: for use as a component within other software tools (e.g. Web search), for insight into other areas of computer science, and to explore the possibilities for managing information. The structure of this chapter will reflect this.

By way of a short introduction, we start by giving a bit of history of database management systems, leading up to the creation of *relational* database management systems, which are the most important class of DBMSs currently. We go on to describe the input languages and structure of a “traditional” (relational) DBMS. We then give a sample of DBR that is oriented towards improving the processing pipelines of existing DBMSs. In a subsequent section, we turn towards research on expanding the functionality of data management systems, covering several significant extensions, sometimes quite radical, that have been explored. In the process we will try to give some idea of how research on these new systems has been integrated into the standard feature set of commercial database systems, and the extent to which it has had influence in other parts of computer science.

Owing to page length limits, for many systems we shall only sketch their main features, giving references for details. We shall devote more space to the informal level and leave technical-level details to references cited. Furthermore, we shall assume that the reader has some familiarity with such basic matters as formatting data on punched cards and magnetic tapes and with common database processes such as *report generators* (just what they sounds like), *sorting* (e.g., sequencing data, say alphabetically, according to some key fields in a table of data) and traditional business applications such as payroll files. These topics are intuitively comprehensible without explicit introductory exposition.

The Relational Model

The Path to Relational Databases

The growth of database management software is a story of the growth of abstraction in computing systems. As in many other areas of computing, in the beginning there were low-level concrete tasks that were carried out first by special-purpose hardware, then by special-purpose programs – for example, reading data from punched cards, performing a hard-coded calculation on the data and then generating results or *reports* in a custom format. Starting from the 1950s there were programs dedicated to processing payroll data, purchase orders, and other pieces of *structured data*. The software was tailored not only to the application at hand, but to the hardware and the input formats. Even the data records themselves, starting with punch cards and later magnetic tape, were often created on a per-task basis, with no sharing of data between applications.

At the end of the 1950s software emerged that abstracted some general functionality used in many data processing activities. Report generation and sorting were two areas of particular interest. This abstract software evolved into the *file management systems* of the 1960s. Although software was still often bundled with hardware, independent software vendors such as Informatics began to offer file-processing software in competition with mainframe vendors like IBM. This decoupling spurred interest in making data processing software independent from hardware, just as later on it would spur independence from the operating system. File managers regarded structured files – plain text files that made use of some set of delimiters – as the basic input abstraction. Here is a description of one system (quoted from Postley 1998).

> This program has been developed in response to a large number and wide variety of requests for reports consisting of selected information from magnetic tape files. These requests usually require the preparation of a new program or modification of an existing program. This program provides a more general solution to the problems of information retrieval and report generation. It combines four generalized capabilities. It can
>
> 1. Utilize any of a wide variety of tape formats
> 2. Make selections on the basis of complex criteria

> 3. Produce reports in a wide variety of list-type formats
> 4. Produce several reports on a single pass of a magnetic tape file. This can be done with no appreciable increase in retrieval time
>
> By providing such a system, it is expected that a reduction in programming and machine time will be realized. These savings, will, of course, be magnified for those retrieval/report generations of short production life and for those reports requiring frequent alterations in selection criteria or report format.

The emphasis even in these "generalized" systems was on batch processing, i.e., on entire files. With the move from magnetic tape to disk storage in the early 1960s, the possibility of querying on-demand emerged. At the same time, the notion that centrally managed data would be a way of radically increasing business productivity became popular within corporate management circles (Haigh 2006). The vision was that managers would have a global integrated view of their business, being able to answer questions "instantly" that would have previously taken hours or even weeks of manual work. The development of advanced decision support and business analysis tools that would realize this vision came much later in the evolution of database systems. But the possibility of such systems inspired corporations to invest more heavily in data processing technology, and gave added impetus to the development of a flexible query language.

Throughout the 1960s systems were developed that had many of the features of modern DBMSs, including some ability to perform querying of shared data and to concurrently process updates sent from multiple processes. IBM's IMS (Patrick 2009) and General Electric's IDS, the latter created by Turing Award winner Charles Bachman, introduced more general procedures for defining data, the precursors of modern data definition languages. While prior systems had left much of the responsibility for management of concurrent updates to application programmers, systems such as IMS managed them transparently. Still, the emphasis was on batch mode or on a fixed set of queries. The view of data provided by IMS upon its initial product release in the late 1960s is described as follows (McGee 2009):

> The data model provided by the initial release of IMS was Data Language/1 (DL/1). In this model, databases consisted of records, records were hierarchic structures of segments, and segments were sets of fields stored in consecutive bytes. One field in a record root segment could be designated the record key. The program's interface to IMS provided calls to access records sequentially or by key; to navigate to segments within the record; and to insert, replace, and delete records.

Although in retrospect the convergence towards the current notion of *data management functionality* is clear, as the 1960s ended, alternative visions of the future were available. On the one hand, overlapping functionality in database systems was incorporated in general-purpose programming languages such as RPG and COBOL. These included querying and data definition as key elements, while still focusing heavily on report generation features that are secondary in a modern DBMS. On the other hand, research projects with far greater scope, incorporating artificial intelligence and natural language interfaces, held out the possibility that data management systems would be subsumed by software with much broader capabilities in the near future.

The current consensus on the functionality of a DBMS arose both from industry trends (e.g., increasing specialization in the independent software industry) and standardization efforts. The Conference on Data Systems Language (CODASYL) was a consortium originally formed to develop a "business language", but which later worked on a variety of computing standards issues, including the development of the business programming language COBOL. In 1965, CODASYL formed a committee later known as the Data Base Task Group (DBTG), which published the first standards for database management systems (Olle 1978). CODASYL arrived at a fairly general definition of the role of data description and data manipulation languages, while proposing concrete interfaces. In general outlines, it resembles that of modern systems.

The deployment of an actual interoperable standard was still quite far off in the marketplace. The functionality proposed by CODASYL was beyond that offered by most database products. Furthermore the underlying model of CODASYL was not that of today's systems. Instead they were based on the *network database model*, an extension of the hierarchic database model used in IBM's popular IMS software. In a network (graph) database, the possible relationships between data entities are fixed as part of the schema – for example, an automobile equipment database might consist of a mathematical graph (see Chap. 7, Appendix) of equipment types, having graph connections (edges) or relationships pointing (for instance) from cars to engines, from engines to cylinders, possibly cycling back to cars. Data is described by graphs (networks) giving the basic kinds of information items and their connections. The basic mode of querying was by navigating these relationships. A query on a database of automobile parts might start by navigating to a particular kind of engine, then moving to an engine component, and then down to a subcomponent. This does represent some abstraction – the description of the data does not deal with particular data formats, or the way that data is represented on disk. But the *navigational* paradigm behind the network model forced the data designer to anticipate all possible relationships in advance. Although the model does not specify how to choose these relationships, many of them would be based on performance considerations – in which direction a querier would be most likely to navigate. Thus the distinction between a data description and an implementation or *index* was muddied. Furthermore the navigational query language forced the query-writer to think somewhat procedurally.

In the late 1960s the *relational model* evolved as a proposed mathematical basis for database management systems. It began with articles advocating the use of mathematical set theory as the core of database query languages. Although there were many forerunners, such as the proposals of David Childs (1968), the model was crystallized in the works of Edgar F. Codd. In his seminal paper overviewing the relational model (Codd 1970), Codd summarized the state of existing approaches to modeling data as follows:

> The provision of data description tables in recently developed information systems represents a major advance toward the goal of data independence. Such tables facilitate changing certain characteristics of the data representation stored in a data bank. However, the variety of data representation characteristics which can be changed without logically

> impairing some application programs is still quite limited. Further, the model of data with which users interact is still cluttered with representational properties, particularly in regard to the representation of collections of data (as opposed to individual items).

Codd defined a simple and elegant underlying model – a definition of what a database is in mathematical terms. He went on to propose that a query language should be employed that allowed users to define any *logical* relationship or access path.

> The universality of the data sublanguage lies in its descriptive ability (not its computing ability). In a large data bank each subset of the data has a very large number of possible (and sensible) descriptions, even when we assume (as we do) that there is only a finite set of function subroutines to which the system has access for use in qualifying data for retrieval. Thus, the class of qualification expressions which can be used in a set specification must have the descriptive power of the class of well-formed formulas of an applied predicate calculus.

Codd proposed two "pure mathematical" query languages – the *relational algebr*a and the *relational calculus* – proving that they had the same expressiveness, and arguing that they could be used as a benchmark by which to judge more realistic query languages. We discuss these languages in the next section.

Codd's work had an enormous impact on database research. It focused attention on the analysis of the behavior of logical formulas on finite structures: evaluation, equivalence, and simplification of logical formulas became a fundamental part of database research. The impact on the database industry was just as large but not as immediate. The presentations of the relational model were written in a highly mathematical style. They were considered unrealistic by database vendors. Thus while the relational *model* was proposed in the late 1960s, relational databases evolved only gradually throughout the 1970s within the research community. A major breakthrough was System R (Chamberlin et al. 1981), developed at IBM research in San Jose. The project was initiated in 1974 and continued throughout the 1970s, with the first customer installation in 1977. The basic features of the relational paradigm – which we describe next – were present in System R, including transaction support, join optimization algorithms, and B-tree indexes. The major database systems of the 1980s, from IBM DB2 to Oracle, descend directly from System R. In the process of creating System R, Donald Chamberlin and Raymond Boyce proposed the SEQUEL language (Chamberlin and Boyce 1974). This evolved into the standard relational query language SQL, the first version of which was standardized by ANSI in 1986.

As the computer industry moved from mainframes and dumb terminals to networks of personal computers, database management systems migrated to the use of a client–server model. In the 1990s with the rise of the Web (see Chap. 7), database systems became an integral part of e-commerce solutions. A database server would typically sit behind a Web server; remote client requests coming via HTTP to the Web server would invoke SQL requests to the DBMS, with results sent back to the client in HTML. By 1999, the relational database market (including OO extensions), was estimated to have revenues of 11.1 billion dollars (an estimate of the market research firm IDC, quoted from Leavitt 2000).

Students

id	*first*	*last*	*credits*
1	John	Smith	123
2	Jane	Doe	44
3	Bill	Wright	67
4	Janet	Who	2

Enrollment

id	*course*	*credits*
1	automata	20
1	databases	40
2	databases	30
3	algorithms	20

Dependencies

course	*dependson*
algorithms	calculus
databases	algorithms
automata	algorithms
web	databases

Fig. 10.1 Example database for the university enrollment setting

We will now review the basic features of the *relational paradigm* – our name for the features of a "traditional relational database", representing both the core of most commercial systems and the view of databases given in most database textbooks.

A Tour of the Relational Paradigm

The relational paradigm is based on several key principles:

- Data abstraction and data definition languages
- Declarative queries
- Indexed data structures
- Algebras as compilation targets
- Cost-based optimization
- Transactions

We will go through the principles individually, using the sample of a university enrollment database given in Fig. 10.1 below.

Abstraction

Abstraction is a key principle in every area of computer science – shielding people or programs that make use of a particular software artifact from knowing details that are "internal". In the database context, this means that users of database systems should be shielded from the internals of database management – what data structures or algorithms are utilized to make access more efficient. Thus a user should be able to define only the *structure* of the data, without any information about concrete physical storage. The interface which describes the structure is a *data definition language*. Access to the data should only refer to the structure given in the definition.

While standard programming languages provide a rich variety of data structures that can be defined by a user, relational languages require the user to describe data in terms of a very simple *table* data structure: a collection of attributes, each having values in some scalar datatype. The attributes of a table are unordered, allowing the data to be returned with any ordering of columns in addition to any ordering of rows.

In the university example, the database creator would declare that there is a table for students, listing their names, their student identifier, and their number of credits. Such a *Students* table with example data is given in Fig. 10.1, along with other tables in the university enrollment database.

The relational database standard language SQL (abbreviating "Structured Query Language", although it contains sublanguages for almost every database task) provides a data definition language in the form of a repertoire of `CREATE TABLE` commands, which allows the user to describe a table, its attributes, their datatypes, along with additional internal information. For example, to create the *Students* table, one could use the following SQL command:

```
CREATE TABLE Students (
   id INT PRIMARY KEY, first TEXT, last TEXT, credits INT)
```

Relational data definition languages allow database designers to describe in a declarative format important aspects of the *semantics* of the data in a way that can be exploited by a DBMS. In particular, they can include *integrity constraints*, which give properties the data needs to satisfy in order to be considered "sane". The `PRIMARY KEY` declaration above states that the *Students* table cannot contain two rows with the same *id* field and is an example of an integrity constraint.

Relational query languages allow the user to extract information according to the structure that has been defined. One could issue a query asking for all students who have taken at least 50 credits of courses. In accordance with the data abstraction principle, queries cannot be issued based on the internals of data storage; a user could not ask for all data on a particular disk, or all data located near a particular item within storage or accessible within a particular data structure.

Declarative, Computationally Limited, Languages

The fact that data is to be retrieved via an abstract descriptive interface leaves open the question of what kind of programming infrastructure could be used to actually access it. One approach would be to use "data-item-at-a-time" programming interfaces, which allow a programmer to navigate through the database in the same way they navigate through any data structure in a general-purpose programming language (Chap. 4): issue a command to get to the entry point of the structure, say an array, and then iterate through it. If we wanted to get the students who have taken at least 50 credits of courses, such an API requires a program (in the host programming language, e.g., C, Java, C++) that issues an API command to connect to the student table, another command to access the first (in arbitrary order) row (or *tuple*) in the table and put it into a host-language variable, and then a loop in the host programming language that performs the following action: checking that the current tuple satisfies the criterion (at least 50 credits) and if so, adding it to the output, then proceeding to the next tuple via calling an API command.

Such an iterative interface is simple for a programmer to use, since it requires only the knowledge of a few basic commands – e.g., a command to get the next tuple and store the result in a local variable. It allows the programmer to exploit all the features of the host language.

There are two main drawbacks to a data-item-at-a-time approach. First of all, it does not give a way for non-programmers to access the data. Anyone who wants to get the students with at least 50 credits has to know how to program. Secondly, and perhaps more importantly, performance will suffer in this approach. The program will have to access every student record in order to access the ones of interest. Further, records will be fetched one at a time, even though the architecture of any computer would allow hundreds if not thousands of records to be transferred between disk and main memory in a single-command.

The alternative pursued in relational database systems is the use of *query languages*; access to the database is by issuing statements that define properties of the set of tuples to be retrieved, giving no indication how they should be obtained. In the example above, a user would state that they would like all student ids for students with number of credit attributes above 50. The SQL representation of this is basically a formal structured version of the natural language phrasing, given in the query Q_0 below:

```
SELECT s.id FROM Students s WHERE s.credits > 50
```

`SELECT` describes a subset of the tuples satisfying the `WHERE` ("such that") clause. The above is a very simple example, but query languages can express fairly complex subset requests. For example, a query asking for the names of students with number of credit attributes above 50 who are enrolled in databases would be the following query Q_1:

```
SELECT s.last FROM Students s
WHERE s.credits > 50 AND s.id IN (
   SELECT e.id FROM Enrollment e WHERE e.course='databases')
```

Here `IN` specifies the set membership relation.

The declarative style of set-theoretic subset interfaces allows additional abstraction: the database manager is now free to choose a procedural implementation of the set theory operations that fits the current storage structure of the data. The second approach has become the ideal for database access and also for database update and transformation – programs or users describe the collection of data items that they would like to see or to change, and the details of how to do this are left to the database manager.

There are many possible declarative languages. Prolog, for example is a paradigmatic declarative language, with no explicit control structures. It nevertheless allows one to express any possible computation, including arithmetic and recursive definitions. Relational database systems, in contrast, looked for languages with limited expressiveness – ones that can only express computations that can be performed reasonably efficiently. The motivation is to prevent users from writing queries that cannot be executed, or queries whose execution will degrade the performance of the database manager unacceptably.

What exactly does *limited expressiveness* mean? At a minimum, it means that queries cannot be expressed in the language if they require time exponential, or more generally super-polynomial, in the database cardinality. In practice, one desires performance much better than this – for large datasets one generally wants implementations that run in time less than $C_1|D|^2 + C_2$ on a database D, where the coefficients C_1, C_2 are not enormous. One coarse benchmark for a query language is given by *polynomial-time data complexity*: for every query Q there should be a polynomial P such that the execution of Q on a database D can be performed in time at most $P(|D|)$. (As usual, $|D|$ is the cardinality of D.)

The standard query language that emerged as part of the SQL standard fulfilled the polynomial time requirement. In fact, a large fragment of the language could be translated into a much more restricted language, a variant of *first-order logic*. (See Chap. 2, Appendix G.) This fragment is the one formed from nesting the basic subsetting `SELECT...FROM... WHERE` clauses of SQL, connecting multiple clauses via the quantifiers `EXISTS` and `NOT EXISTS`, or (equivalently) with the set membership constructs `IN` and `NOT IN`. We refer to this fragment as *first-order SQL* in the remainder of this chapter. For example, the following first-order SQL query Q_2 retrieves the names of students who did not take databases:

```
SELECT s.last FROM Students
WHERE s.id NOT IN (
  SELECT e.id FROM Enrollment e WHERE e.course='databases')
```

First-order SQL translates to a simple syntactic variant of first-order logic known as *relational calculus*. Every relational calculus query can be performed in polynomial time on a Turing machine, and in constant time on a parallel machine. The translation to a predicate logic is not used for compilation of the language. (As we shall see, queries are compiled into algebraic formalisms instead.) But predicate logics are often useful for reasoning about the properties of a query language, since logics are well-understood and well-studied formalisms. There are translations in the other direction as well: for example, SQL can express all Boolean queries that can be defined in first-order logic over the relation symbols. While the polynomial time requirement represents a limit on the expressiveness of query languages, the requirement of expressing all queries in a logic gives a lower bound on expressiveness, often referred to as *relational completeness*. The ideal would be to have a query language that corresponds in expressiveness exactly to a predicate logic – preferably one with a well-established set of proof rules. Then the optimization rules of the query language could be justified by the soundness of the proof system for the logic. Relational languages do not meet this ideal – SQL is much more powerful than first-order logic, and does not correspond exactly to any well-studied logic – but they approximate it.

Just because database query languages are limited in expressiveness does not mean that users are restricted in performing certain tasks. For example, a user can still filter data based on some complex arithmetic comparison or recursive function. The idea is not that query languages would replace general-purpose programming languages. They would only be used to express requests for information that requires searching and combining data from a large dataset. Finer filtering of information would be performed within a general-purpose language.

An example of the runtime flow for our first query Q_0 might be:

```
$Q_0 \leftarrow$ SELECT s.id FROM Students s WHERE s.credits $>$ 50;
$D_0 \leftarrow$ newDatabaseConnection();
$results \leftarrow D_0$.execute($Q_0$ );
while not end of $results$ do
    $studentRecord \leftarrow results$.next();
    if good($studentRecord$.credits) then print($studentRecord$ );
endw
```

The "execute" operation evaluates the query Q_0 on the stored data. The result of execution may be the transferring of all of the data into memory, or just the determination of the initial record satisfying the query, with the remaining records pulled in on demand. The "next" operator iterates through all of the records satisfying the query. The filter "good" is a function on tuples written in the host language, and could use any features available in that language. A database management system thus divides up work between the host programming language and a special-purpose language.

Indexed Data Architecture

Relational database managers were designed for datasets that would be too large to fit into a computer's main-memory. At any point in time, a portion of the data would be in memory (in a buffer cache) and this portion could be accessed and navigated quickly. The remainder would be on secondary storage (e.g., disk drives). The disk-resident data can be divided up into *blocks*, a unit that can be transferred to main-memory in one atomic operation. Query processing would involve locating relevant blocks of data on disk, transferring block by block to the buffer, and then locating the required data items by navigating a block.

The main tool relational database managers use to speed query processing is the maintenance of auxiliary data structures that allow retrieval with fewer accesses. The principal example of this are tree indexes, such as B-trees and B^{+-}trees. In the student example, we might create an index on *id*. If the ids range among 8-digit numbers, the first level of the tree divides these numbers into some number of intervals, and similarly each of these intervals is split into subintervals in the next node.

To find the student with id 12345678, we follow a path down the tree by locating 12345678 within the collection of intervals under the root, then within the collection of subintervals, until finally arriving at the leaf node containing the block where the student record resides. Since the internal nodes of a tree index contain only a subset of the values for one attribute (hence only a subset of the ids), they will generally be dramatically smaller than the dataset. Still for a large dataset, the bulk of the tree would reside on disk.

The trees used are *balanced*, like many tree structures used in computing: the maximum number of levels below any given node is thus fixed, and this guarantees that the access time for an *id* value will not vary from id to id. As data is modified the tree indexes must be updated, but standard tree update algorithms can be applied to make the update time a constant factor (between 1 and 2) in the number of updates.

The key thing that distinguishes tree indexes from other tree data structures is their *branching*. Instead of using binary branching, as search trees elsewhere in computing do, the branching in B-trees is chosen so that one internal node can be stored in a single block of memory, and thus a single navigation step in a tree requires at most one data transfer step from disk to memory. Depending on the block size of the machine and the size of a data item, there might be hundreds or even thousands of entries at a given level.

Algebraic Query Plans

One of the key advantages of declarative languages is that the optimizer can choose the best implementation, using a more global view of the query than the compiler for a data-item-at-a-time language would possess.

A way to capture this extra dimension of flexibility between queries and evaluation mechanisms is via the notion of a *query plan*. A *plan* is a description of high-level steps that implement the query. Many plans can correspond to the query, and have different performance characteristics.

Consider again the query Q_1 above. A naïve query plan would correspond to the following step: getting all ids of students taking the course "databases", using an index I on the table, reading the blocks of this result one at a time; finding all the student records that correspond to these ids; scanning through them to check the number of credits, returning only those above the credit threshold; scanning through the list of student records that survive the filtering process and returning all the name fields. This plan might be represented internally within a database manager by the following expression:

$$\pi_{last}(\sigma^{scan}_{credits>50}(Students \quad \bowtie \quad \pi_{id}(\sigma^{I}_{course=''databases''}(Enrollment)))).$$

Here, $\sigma^{I}{}_{course=''databases''}$ refers to a *selection* operator, which uses index I to retrieve all records on a particular course; $\bowtie$ is a *join* operator, which takes two tables and merges all matching records – in this case, a table of ids and a table of student records; $\sigma^{scan}_{credits>50}$ is an operator that selects students within a student table above 50 credits, via just iterating through the table block by block; finally, π_{last} and

π_{id} refer to *projection* operators, which remove all columns from a table except, respectively, the columns *last* and *id*, eliminating duplicate rows.

Of course, many details are omitted from this plan; in particular, there are many ways of implementing the join operator $\bowtie$. This plan follows the structure of the query, and thus represents a fairly naïve implementation. The key point is that there are many other plans that implement the same query, some of which will not follow the structure of the original query closely. For example, another plan is as follows: use the index on the *Enrollment* table to get the list of records for "database", then use another index *J* on the *Students* table to get all student records for students having at least 50 credits; then join the two tables; finally, remove all but the *last* field, eliminating duplicates. This plan might be represented as follows:

$$\pi_{last}(\sigma^{J}_{credits>50}(Students) \;\bowtie\; \sigma^{I}_{course=''\mathrm{databases}''}(Enrollment)).$$

The plan expressions that we are displaying are in a language called the *relational algebra*. The *relational algebra* is still declarative. It has the same advantage over formalisms such as the relational calculus as compilation formalisms for general-purpose programming languages have over the corresponding source languages: they are easier to optimize because there are fewer syntactic operators. In particular, the language is *variable-free* – there are no explicit variables in the syntax – and thus the conditions under which a new query can be formed from composing a new operator are simpler.

The process of getting from a query to an efficient plan expression consists of a translation to algebra and then transforming via applying equivalences – analogous to the application of algebraic rules such as commutativity and associativity in algebra. The standard example of such a rule is *pushing selections* inside projections or joins: a query plan

$$\sigma_{course=''\mathrm{databases}''}(\sigma_{credits>50}(Students) \;\bowtie\; Enrollment)$$

would be converted to

$$\sigma_{credits>50}(Students) \;\bowtie\; \sigma_{course=''databases''}(Enrollment).$$

In searching through plans by applying transformations, we are exploring the space of possible implementations of the query.

Cost Estimation and Search

The translation to algebra and the use of transformation rules allows one to explore the implementation space. But two issues remain: how does one determine how efficient an implementation is? And given that the search space is large – indeed, the collection of equivalent expressions is infinite – how does one search through it in a way that makes it likely to find the best plan?

Relational database managers approach the first question by defining heuristic cost estimates on a per-operator basis. Of course the real cost of basic operations, such as retrieving all elements that satisfy a given selection criterion, depends on the data – one cannot know it exactly without executing the query. Statistical information about the data, refreshed periodically, can provide a substitute for exact information. For example, if one stores a histogram telling what percentage of the students have credit totals in any interval of length 5 between 0 and 150, then one can get a very accurate estimate of the number of students having above 50 credits. This will allow one to estimate the cost of the selection on *I* in both plans above. Cost estimation of basic operations on relations is highly tied to the index structures and physical storage – it takes into account index structures, when they are present, cached data, and locality of data on disk. Thus much of the implementation of relational structures is encapsulated within a cost function, which serves as an interface to the query optimizer.

In terms of the second question, relational database managers have no universal solution for searching the space of query plans. They apply some standard search techniques, but customized to the database setting. In particular, they rely heavily on divide-and-conquer, breaking up the algebraic expression into subparts and optimizing them separately; the variable-free nature of relational algebra expressions makes it easy to analyze components of queries in the same framework as queries, which assists in defining algorithms via recursion on query structure.

ACID Transactions

Above we have focused on querying databases, but databases are also being updated concurrently with query accesses. The concurrency of updates and queries introduces many issues. Consider two users of our university database. User A is doing an update that is removing a student S from the *Students* table along with all of the student's records in the *Enrollment* table. Concurrently user B is querying for the average number of courses for any student, a query that involves both the number of total courses in the *Enrollment* table and the number of students. The high-level query of B translates to a number of access operations on the database, while the update performed by A translates to changing records in two distinct tables. If the low-level operations are interleaved in an arbitrary order at the database, then the users may see anomalous results: the average seen by B might reflect a student table that includes student S, but an *Enrollment* table that lacks the records of S, or vice versa. Indeed, the average seen by B might reflect a table including only a portion of the enrollment records of S.

The issue is related to the level of abstraction provided to users by a database system; a complex data-intensive activity like querying for an average is provided as a single primitive to user B, who will consider it to be *atomic* – something that cannot properly overlap with other database activities. At the very least, the database should support this. Furthermore, user A would like an even higher-level of abstraction: A would like the two updates together to be considered as a single primitive, which should not properly overlap with other database activities. A DBMS provides additional language support that allows A to do this – to state that the delete of student S and the elimination of S's enrollment records represents a *transaction*, which should have the properties of a single indivisible action.

Above we have spoken of an action or sequence of actions being treated as "atomic" or "indivisible". But what does this mean in practice? Relational databases have formalized this by the requirement that transactions satisfy the following properties:

Atomicity
No transaction should be "implemented in part". If there is a failure in the process of performing one of the updates in the transaction (e.g., due to hardware failure or integrity constraint failure), then any other updates that have been applied should be *rolled back*, and their effects should not be seen by other database users.

Consistency
Transactions should leave the database in a consistent state: one in which all integrity constraints hold.

Isolation
Until a transaction has completed, no concurrent user should see results that are impacted by the updates in the transaction.

Durability
Conversely, once a transaction has completed, its results should not be rolled back regardless of hardware or software failures. We say that the transaction should be durable.

Support for transactions that satisfy the properties above – abbreviated as *ACID transactions* – is one of the main goals of relational systems. ACIDity can certainly be achieved by running each transaction serially: assigning each one a timestamp, and running the transactions in timestamp order (queuing those with lower timestamp), rolling back the transactions that do not complete. Logging mechanisms can be used to track the impact of the transaction to enable rollback. The problem is that this can lead to unacceptable delays in running updates and queries. The goal is then not just to enforce the ACID properties, but to enforce them while allowing updates and queries to proceed without blocking whenever this would not destroy ACIDity – that is, to allow as much concurrency as possible.

Locking mechanisms are the most popular technique for managing concurrent use of the data; transactions are only allowed to modify data items that are not locked by another transaction, and when they act on an item they receive a lock on it, which generally remains in place until the transaction completes or aborts. When a transaction queries data it receives a weaker lock on the data, one which

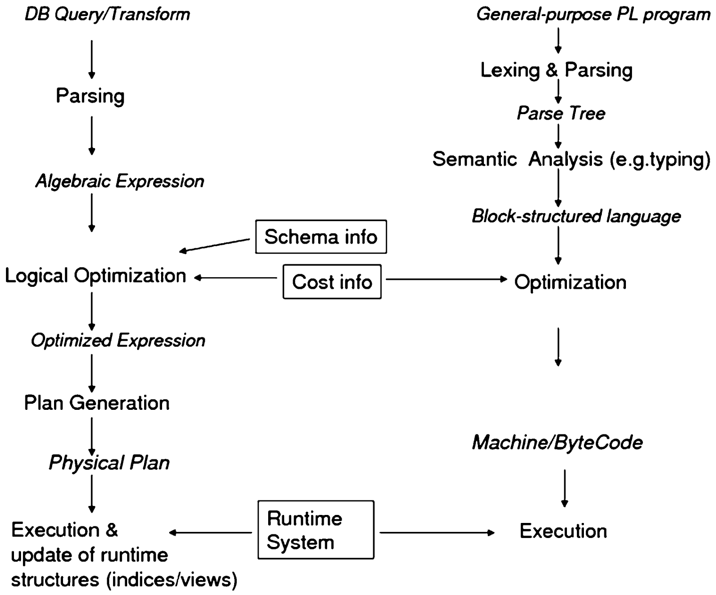


Fig.10.2 Processes for database programs and general-purpose programming languages

allows other transactions to query the same data but not to update it. Lock-based concurrency control has many variants, particularly concerning the granularity at which data is locked.

The Evaluation Pipeline of the Relational Paradigm

The relational paradigm gives a flow of processing for queries that is closely-modeled on the flow of processing of programs in a general-purpose programming language (GPPL), such as C or Java. Figure 10.2 shows a comparison of the processing flow at query/program evaluation time for a general-purpose program and for a database query or transform. In both cases, source language expressions are parsed, and eventually arise at a form more suitable for optimization.

The algebraic expressions are run through a *logical optimizer* which has an abstract interface to information about program executions. The optimized expression is either translated directly into an executable plan, or is run with the help of a runtime system. In the case of a DBMS, the runtime system would include indexes and other auxiliary data structures.

Several differences stand out between the two scenarios:

- Database languages are much smaller and syntactically less complex than GPPLs, and hence the parsing stage is fairly uninteresting. Similarly, the semantic analysis phase is often much simpler than for a GPPL.

- Database queries are often issued interactively, or sent from across a network. Thus optimization as a rule must usually be quite a small phase – in seconds if not milliseconds. In the case of GPPLs, the common case is that the program text is available for some time before execution, making a more robust optimization phase possible.
- GPPLs often make use of interesting runtime data-structures – byte-code interpreters, or garbage-collectors. But not all GPPLs do, and one generally cannot say that a particular program requires auxiliary runtime structures. In most applications database queries could not be executed at all without the help of large and complex runtime structures. These structures, in turn, need to be maintained after program execution (see bottom left in the figure), which may be as complex and as time-consuming as execution itself. Designing and maintaining these structures thus plays a central role in DBR.

  Related to the above, DBMS systems are often tuned for many runs of a query or set of queries, not just for an individual execution. The initial population and period maintenance of runtime structures may thus represent an independent process, taking place far prior to execution or at intervals between executions.
- A DBMS makes use of not only a program text, but information about the semantics of the data – the schema. There is no corresponding standard input description for GPPLs.
- A distinction that is not exhibited in the figure, but has an even greater impact, is that, typically, database programs run in a heavily concurrent environment, with hundreds, even tens of thousands of concurrent queries and updates sharing the same data. Hence the handling of concurrency is paramount.
- Both GPPLs and DBMSs can be used in a distributed setting where resources (data, computing power, etc.) are distributed over a computer network. However, for reasons that we will explain further on, even when there is no inherent distribution in these resources, it is common for a DBMS to manage its data in a distributed manner.

Let us return to the question of the manifold nature of database research mentioned in the introduction. We said that part of database research is driven by improving the performance of existing database management products, while other DBR is geared towards exploring the possibilities for managing data and extending the use of database management techniques.

If we consider the first kind of research, parallels with programming languages suggest that its structure could be broken down along the same lines as for PL research – there is research on optimization, research on improving runtime data structures, concurrency, etc. Of course, as the figure shows, DBR in no sense reduces to PL research, and indeed the research on DB performance has not been closely-tied to work in PL. Still in *rough* analogy with programming language work, one would expect the structure of DBR to follow the lines of the flow on the left-hand side of Fig. 10.2. And indeed much of the first kind of database research naturally works in exactly this way. We will refer to this as *core database research*, and we will give an idea of some aspects of it in the next section, following the processing flow on the left of Fig. 10.2 in our tour.

For research extending the functionality of DBMSs, and considering ways in which data *could* be managed, there is a dichotomy: some of the work tries to preserve the flow of processing in the relational paradigm, but with some new functionality at the query language level. Some of this work takes as a given the relational perspective of database query languages as a logical formalism, and looks at to what extent other logics could be evaluated in the same way as first-order predicate logic on relations. A second line in the more speculative kind of DBR looks at more radical changes in the processing pipeline, which have no analog in programming languages. We will overview both of these extensions further in this chapter.

Core Database Research Sampler

In this section, we provide a sampler of what research in the database field has focused on and accomplished over the years concerning the processing pipeline of relational database management systems, as was presented in the previous section. Following Fig. 10.2, we start at the query level, then move to logical optimizations, physical optimizations, down to the execution of the query on actual hardware. Each research area is represented by a selection of significant research results, with no intention of exhaustiveness or objectivity in the choice. We also indicate the impact research has had on the design of modern DBMSs, discussing whether models, algorithms, and data structures from the scientific literature have been implemented in widely used systems.

Query Languages

We previously explained how first-order SQL has nice logical and algebraic interpretations in terms of relational calculus and relational algebra. However, even the very first version of the SQL query language (Chamberlin and Boyce 1974) went beyond that fragment and included *aggregate functions* and *grouping*. Over the years, the standard and implementations of the SQL query language evolved towards more and more expressive power. We present in this section the additions that have been made to SQL to overcome some of its limitations, both in the standard and in actual implementations, that do not always follow it strictly.

Aggregation

One of the most common functions of database management systems is to compute summaries of existing data by aggregating the numerical values that appear in these tables. Going back to the example database of Fig. 10.1, one can for instance ask the following questions:

- What is the average number of credits obtained by students?
- How many students have more than 50 credits?
- What is the maximum number of credits earned by a student enrolled in the databases course?
- For each course, what is the average number of credits of students enrolled in this course?

All these queries make use of an *aggregation function* to compute the average, count, or maximum of a collection of values. The last query also uses a *grouping operator*, where the aggregation is performed for each group of results to a subquery that have the same value for a given attribute. As already mentioned, both aggregate functions and grouping are basic features of SQL. They correspond to forming families of sets.The last query can for instance be expressed as:

```
SELECT e.course, AVG(s.credits)
FROM Students s, Enrollment e
WHERE s.id = e.id
GROUP BY e.course
```

It is also possible to define extensions of the relational algebra for queries that involve aggregation and grouping (Klug 1982; Libkin 2003). Using a similar notation as in Libkin (2004), a relational algebra expression for the query above is:

$$Group_{course}[\lambda S.\mathrm{Avg}(S)](\pi_{course,credits}(Students \bowtie Enrollment)).$$

As in the non-aggregate case, database management systems use such algebraic expressions to represent and manipulate query plans.

In essence, the queries definable in the early versions of SQL (Chamberlin and Boyce 1974; ISO 1987) are the ones of this aggregation and grouping algebra. Queries with aggregation and grouping, with their standard syntax and semantics, are supported by all relational database management systems. In the following, we will refer to this language as *vanilla SQL* to distinguish it from more recent and less well-supported additions.

Recursion

A natural question is that of the expressive power of vanilla SQL. Can all "reasonably simple" queries that one may want to ask over a relational database be expressed in SQL? Again, consider the university enrollment example. The table *Dependencies* lists the courses that a student must have followed in the past in order to get enrolled in a given course. Suppose that a new student aims at taking the Web course. Then he needs to query the database to retrieve all courses this one depends on, so as to plan his curriculum. This is a simple enough request: if the table *Dependencies* is seen as the relation defining a directed graph, the problem becomes determining all nodes in the graph reachable from the "web" node. This can be solved in time linear in the size of the relation by simple depth-first or breadth-first graph search algorithms (in other words, by computing the transitive closure of the relation); because of the inherent recursion in these algorithms, such a query is called *recursive*. We have isolated earlier in the chapter polynomial-time data complexity as an indicator of the limits of expressiveness for database query language. The course dependency query and other similar recursive queries fit this criterion. Are they expressible in vanilla SQL?

It is well established that recursion cannot be expressed in first-order logic (and thus, in the relational calculus). This can be proved using a *locality* (Libkin 2004) argument: a relational algebra expression is unable to distinguish between two nodes in a graph whose neighborhood of a certain radius are isomorphic, while computing the transitive closure is essentially a non-local operation. It turns out that the same result holds for vanilla SQL (Libkin 2003), with aggregation and grouping, when datatypes are unordered (the problem is more complex and still open for ordered datatypes).

The (apparent) inability to write simple recursive queries in SQL has led the designers of the SQL3 standard (ISO 1999) to add to the language the `WITH RECURSIVE` feature that enables recursion. As an example, the course dependency query can be written as:

```
WITH RECURSIVE Closure(course) AS (
   VALUES('web')
   UNION
   SELECT d.dependson FROM Dependencies d, Closure c
   WHERE d.course = c.course
) SELECT * FROM Closure
```

Support for this kind of query in DBMSs varies. In Oracle, for instance, it is not possible to use `WITH RECURSIVE` queries at the time of this writing. A similar feature, however, has been available in Oracle since the early 1980s (Stocker et al. 1984) with the proprietary `CONNECT BY` operator, which demonstrates the early interest in such a functionality. IBM DB2, Microsoft SQL Server, and PostgreSQL all support `WITH RECURSIVE`, while other less feature-rich DBMSs such as MySQL do not allow any form of recursive queries, short of stored procedures.

Stored Procedures

The components of the SQL query language mentioned so far (the relational algebra, aggregation and grouping, recursion) all have in common a polynomial-time data complexity. As already discussed, this is a design choice, to avoid queries that would be too costly to evaluate. The philosophy was that more complex processing of the data would be done outside the database management systems, in applications written in traditional programming languages. This may mean, however, redundant implementation of data-related functionalities (e.g., data validation) in all applications (possibly written in different programming languages) that interact with a given database. For this reason, users have felt the need to move larger parts of the application logic, in the form of arbitrary code that manipulates data, into the database management system. Database vendors have thus offered the possibility to implement *stored procedures* and *user-defined functions* directly in the DBMS, using extensions of the SQL language with control flow statements (variable assignment, tests, loops, etc.). These procedures are stored in the database itself, along with the data. This has led in turn to the addition of stored procedures to the SQL standard, under the name SQL/PSM (ISO 1996) (for *persistent stored modules*). Though few vendors follow this standard to the letter and there are many variations in the actual stored procedure languages used in DBMSs, all major systems provide this functionality. Oracle's stored procedure language, PL/SQL, introduced in 1992, has been especially influential.

Using stored procedures, it is possible to implement arbitrary processing of the data inside the database management system (in other terms, the addition of stored procedures make the SQL language Turing-complete). As a consequence, queries making use of stored procedures do not have any guarantee of polynomial-time data complexity and most query optimization techniques are not applicable any more. The database management system focuses on optimizing subparts of the stored procedures that do not make use of control flow statements.

The relational algebra, aggregation and grouping, recursive queries stored procedures and user-defined functions are the tools that modern database management systems provide to query relational databases. Other commonly available features of the query language either add syntactic sugar on top of these basic functionalities, or allow the querying of other kinds of data structures, such as XML documents (see further), geospatial coordinates, or plain text queried through keyword search.

Logical Optimizations

Let us move to the realm of query optimization. The goal here is to find an efficient way of evaluating a user's query. As already mentioned, database management systems do this in two ways: by first rewriting the query into a form that is easier to evaluate, independently of the data it runs on, and then by generating a set of possible evaluation plans for the query and using statistical information to choose an efficient one for this particular database. We are looking now at the former type of techniques, that we call *logical optimizations*. Plan generation and statistics-based cost estimation are described further. Logical optimizations can either be *local* (the query is rewritten parts by parts) or *global* (an optimal rewriting of the query as a whole is sought for). Since optimizations considered here are independent of the actual data they are particularly useful when the same query is run multiple times over different database instances.

Local Optimizations

For reasons that we shall attempt to explain further on, though research has considered and proposed both local and global logical optimization strategies, DBMSs mostly use local optimizations. To go beyond what is presented in this section, a good starting point is Chaudhuri (1998).

Equivalence Rules

The first idea used for query rewriting has already been mentioned: exploiting equivalence rules of relational algebra expressions (especially, commutativity or distributivity of operators). Thus, it is often more efficient to push selections inside joins, i.e., evaluate the selection operator in each relations before joining two relations, or to distribute projection over union, i.e. to transform $\pi_A(R_1 \cup R_2)$ into $\pi_A(R_1) \cup \pi_A(R_2)$. It is not always clear, however, when applying a given equivalence rule makes the rewritten query more efficient. Therefore equivalence rules are also used extensively for generating the space of query plans a cost-based estimator chooses from. Equivalence rules of relational algebra expressions involving classical operators are folklore, but each time a new operator has been considered, new equivalence rules have been investigated. This is the case, for instance, in Rosenthal and Galindo-Legaria (1990) with the *outer join* operator that retains every tuple of one of the two tables being joined, even if no matching tuple exists in the other table. Equivalence rules are an important component of the query optimizer of all DBMSs.

Unnesting Complex Queries

Another form of logical optimization at a local level deals with *nested* SQL queries. SQL offers the possibility of expressing complex queries that use nested sub-queries, especially in the WHERE clause:

```
SELECT s.first, s.last
FROM Students s
WHERE s.id IN (
  SELECT e.id FROM Enrollment e WHERE e.course='databases')
```

A naïve evaluation of this query, which asks for names of students enrolled in the “databases” course, would enumerate all tuples of the *Students* table, and, for each of them, would evaluate the sub-query and return the tuple if the sub-query returns the identifier of the student. Obviously, in this particular example, such complex processing is not required, since the query and its sub-query are *uncorrelated*:

the sub-query does not refer to the current tuple of the main query. This means the sub-query can be evaluated once and its results used for matching identifiers from the *Students* table.

There are more complex examples of nested queries, for which such a simple strategy cannot work. Consider for instance the following query, that retrieves names of students enrolled in a course that would allow them to graduate (assuming they need 150 credits to graduate, counting those they were already awarded):

```
SELECT s.first, s.last
FROM Students s
WHERE s.credits < 150 AND s.id IN (
   SELECT e.id FROM Enrollment e
   WHERE e.credits > 150 − s.credits)
```

Here, the main query and its sub-query are *correlated*: the sub-query has a condition on the value of the current tuple of the *Students* table.

Research has investigated the conditions under which nested queries could be unnested and rewritten as simple one-block queries. The seminal work (Kim 1982) proposes an algorithm to simplify nested queries, which depends of the kind of correlations existing between a query and its sub-query. In this example, a simplification is possible, and yields:

```
SELECT s.last
FROM Students s, Enrollment e
WHERE s.id = e.id AND s.credits < 150
   AND e.credits > 150 − s.credits
```

This rewritten query can be shown to be equivalent to the original one. This *decorrelation* procedure is actually very simple when the two tables are joined by the `IN` operator and when the correlation between the two tables does not involve any aggregate function: just put all conditions of the sub-query in the main `WHERE` clause, and replace the `IN` operator with an equality join. Other works, such as Dayal (1987), have extended the decorrelation procedure of Kim (1982) to support, for instance, grouping, and other forms of correlation between a query and its sub-query.

Obviously, whenever such simplifications are possible (which is not always the case!) they can be applied repeatedly to complex sub-queries to reduce the number of nested blocks, possibly reducing them to a simple `SELECT-FROM-WHERE` query. The reduced query is typically more efficient to directly evaluate, leads to a search space of query plans of reduced size, and is also more easily subject to other forms of logical optimizations, such as static analysis. For these reasons, modern-day query optimizers perform such unnesting, at least in simple cases (Lorentz 2010).

Global Optimizations: Static Analysis

We discuss here a global approach to query optimization, based on *static analysis*: independently of actual data, determine important characteristics of a query as a whole that can be used, in particular, to determine whether the query can actually return any result, what is the optimal evaluation order, or how to rewrite it into a simpler one.

We limit ourselves in this section to the well-understood case of *conjunctive queries*, the fragment of the relational calculus without disjunction, negation, or universal quantifiers. A conjunctive query is thus a conjunction of relational facts (also called *subgoals*) involving either output variables, or existentially quantified variables, or constants, for instance the query Q_1, previously introduced:

$$Q_1(last){:=}\exists i \exists f \exists c.Students(i,f,last,c) \wedge c{>}50 \wedge Enrollment(i,''databases'').$$

Equivalently, this can be seen as a fragment or the relational algebra with the selection, projection, join, and cross product operators, or also as simple SQL queries involving only the `SELECT`, `FROM`, and `WHERE` keywords.

The most basic static analysis problem is that of *satisfiability*: does there exist a database for which the query returns a non-empty result? In the case of conjunctive queries, and without any restrictions on the data, it is easy to see the answer is always yes, because of the monotonicity of the query language. For the full relational calculus, satisfiability is undecidable. The undecidability of the existence of arbitrary models of a first-order logic formula is a consequence of Gödel's incompleteness theorem (see Chap. 2, Appendix G); however, the proof of the undecidability of relational calculus satisfiability (Di Paola 1969) relies on the undecidability of the existence of *finite* models of a first-order logic formula, a result known as Trakhtenbrot's theorem (Trakhtenbrot 1963).

Query Evaluation and Query Containment

Another fundamental problem in static analysis is *query containment*: query Q_1 is said to be *contained* in query Q_2 ($Q_1 \subseteq Q_2$) if, for all databases D, the set of results of Q_1 over D, $Q_1(D)$, is a subset of $Q_2(D)$. For conjunctive queries, a fundamental result known as the *homomorphism theorem* (Chandra and Merlin 1977) relates query containment and query evaluation through the *canonical database* D^Q of a conjunctive query, constructed as follows: each subgoal occurring in the query forms one tuple of its canonical database, and each constant occurring in the query or output variable of the query is the sole tuple of a new unary relation. A *homomorphism* between two relational databases is a mapping h from one to the other such that if $R(c_1 \dots c_n)$ is a tuple of the first database, then $R(h(c_1) \dots h(c_n))$ is a tuple of the second. The homomorphism theorem states that the following three statements are equivalent:

1. $Q_1 \subseteq Q_2$;
2. The output variables of Q_1 are in $Q_2(D_1^Q)$;
3. There is a homomorphism from D_2^Q to D_1^Q.

In other words, testing containment amounts to evaluating a query, and, conversely, evaluating a query Q over D amounts to testing containment of a query for which D is the canonical model in Q. An efficient algorithm for query containment would thus give us an efficient query evaluation strategy. It is easy to see, however, that conjunctive query containment is an NP-complete problem (Chandra and Merlin 1977), and (thus) that in terms of *combined complexity* (Vardi 1985) (i.e., the complexity when both the data and the query are part of the input), query evaluation is NP-complete. In fact, satisfiability of propositional formulas in conjunctive normal form (Chap. 2, Appendix 1), the most well-known NP-complete problem, (see Chap. 13), is a special case of query evaluation over a database where the domain of each relation has only two elements.

Acyclic Queries

Given this intractability of conjunctive query evaluation in the query size, research on query optimization has investigated subclasses of conjunctive queries for which (a) containment is in polynomial time and evaluation is therefore in polynomial time in combined complexity and (b) the recognition problem – determining whether a query belongs to the subclass – is tractable. In particular, the class of *acyclic* queries (Chekuri and Rajaraman 2000) has been widely studied.

To define acyclicity, first consider a simple case when all relations used in a conjunctive query have arity 1 or 2. We define the graph of such a query. Nodes of the graph are constants or variables occurring in the query, and there is an edge between two nodes if there is an atom that involves both these nodes. A query is acyclic if its graph is acyclic. For example, the query $\exists x \exists y \exists z R(x,y) \wedge R(y,z)$ is acyclic, while $\exists x \exists y \exists z R(x,y) \wedge R(y,z) \wedge R(z,x)$ is cyclic. For general conjunctive queries with no restriction on the arity of the relation, the definition of acyclicity involves the hypergraph of the query, and essentially means that there exists a tree-like decomposition of the hypergraph; the precise definition is a bit technical and can be found in Beeri et al. (1981), along with equivalent characterizations.

Acyclic queries are of particular interest because query evaluation can be performed in polynomial-time combined complexity (Chekuri and Rajaraman 2000). Intuitively, for a query with no output variables, it is possible to use the tree decomposition of the query as a query plan where join operators can be replaced with *semijoins* (the semijoin of two relations is the tuples of the first relation for which a matching tuple exists in the second one). One can easily see that testing acyclicity is also tractable. Numerous queries encountered in practice are indeed acyclic; this is the case of most queries related to the university enrollment example encountered so far. But some simple queries are cyclic. Consider for instance the following one, that checks if there is any student enrolled at the same time in a course and one of its prerequisites:

$$\mathrm{Q}_3 := \exists i \exists c_1 \exists c_2 . \mathit{Enrollment}(i, c_1) \wedge \mathit{Enrollment}(i, c_2) \wedge \mathit{Dependencies}(c_1, c_2).$$

This query is obviously cyclic. In order to extend tractable query containment to simple yet cyclic queries, the notion of *treewidth* (Chekuri and Rajaraman 2000) and the more general *hypertree-width* (Gottlob et al. 2002b) have been introduced to characterize the "degree of cyclicity" of a query; intuitively, the tree decomposition of a query is allowed to have more than one subgoal, and the maximum number of these subgoals in the tree gives the width. We omit the precise definition of these concepts, but treewidth and hypertree-width have in common that acyclic queries have width of 1, a "cycle" query such as Q_3 has width of 2, and, more generally, the more complex the sharing patterns of variables across subgoals, the larger the width. Though computing the treewidth or hypertree-width is NP-hard, there are polynomial-time algorithms that check whether a query has width less than or equal to a given constant k. Furthermore, if a query has treewidth or hypertree-width at most k for any fixed k, query containment can be tested in polynomial time, and thus query evaluation is polynomial-time in the size of the query and data. The advantage of hypertree-width over treewidth is that for some queries, the hypertree-width can be arbitrarily smaller than the treewidth.

Query Minimization

A query optimization problem orthogonal to finding an efficient evaluation strategy is *minimization*: does the query have a minimum number of subgoals among all equivalent queries, and if not, how can we rewrite it into an equivalent, minimal, query? As a rule of thumb, the shorter a conjunctive query is, the faster it can be processed. The problem of query minimization is especially significant when the query is not hand-written but automatically generated, e.g., by a content management system or in data integration contexts. Such automatically generated queries are commonly much longer than minimal equivalent queries, involving many join computations that could be avoided.

A consequence of the homomorphism theorem is that every conjunctive query has a unique minimal equivalent query (up to the renaming of variables). Furthermore this minimal query is a homomorphic image of the original query. A strategy for query minimization (Abiteboul et al. 1995) is thus to repeatedly try reducing the overall number of subgoals of the query by mapping variables to constants or other existing variables, testing at each step whether the reduced query is still equivalent to the original query. When no further reduction in the number of subgoals is possible, we have obtained the minimal query. Consider for instance the Boolean query $\exists x \exists y \exists z . R(x, y) \wedge R(z, 5)$. By mapping x to z and y to 5 we obtain the reduced query $\exists z . R(z, 5)$ which can be checked to be equivalent to the original query. Further minimization is obviously impossible, so this is the minimized query.

Note that the homomorphic image of an acyclic query is also acyclic. This means that this minimization procedure can be run in polynomial time over acyclic conjunctive queries. A query optimizer can thus start by checking whether a given query is acyclic, and if so, minimize it, to reduce the cost of its evaluation, and also evaluate it in polynomial time. Once again, this approach can be generalized to queries of bounded treewidth or hypertree-width.

Use of Static Analysis in DBMSs

Basic static analysis is used in relational database management systems, e.g., to test if a query is acyclic in order to replace joins with semijoins (Lorentz 2010). More advanced treewidth-based query evaluation and query minimization, to the best of our knowledge, are not used in any major database management systems, despite the potential usefulness of such techniques in practical applications (Kunen and Suciu 2002). The reason may be that queries are often already minimal and in an easily evaluable form when hand-written; more and more scenarios call for automatically generated queries of arbitrary structure, however. Another reason is the generally good performance of classical cost-based optimizers.

Between Logical and Physical Optimizations: Views

Before moving down the query processing pipeline to plan generation and cost-based query optimization, let us remain at the logical level to discuss *views*: a *view* is a named query that can be used in other queries as if it were a base table in the database. Views can be defined in SQL like this:

```
CREATE VIEW PredictedCredits AS
  SELECT s.id, s.credits + SUM(e.credits) AS credits
  FROM Students s, Enrollment e
  WHERE s.id = e.id
  GROUP BY e.id
```

This statement creates a view *PredictedCredits* that contains the number of credits for each active student at the end of the term, provided they pass all courses they enrolled in. This view can now be referred to in subsequent queries (e.g., `SELECT AVG`(*credits*) `FROM` *PredictedCredits*).

Views can either be *virtual* or *materialized*. Virtual views are just aliases for subqueries; when they are used inside a query, they are substituted with their definition. The full expansion of queries that use views can become relatively complicated, and the unnesting procedures discussed in the previous section can be helpful to optimize them. In materialized views, the situation is different: the query defining the view is evaluated to produce the result table, and this result table is stored and can be directly used as a table in query evaluation.

Virtual views can be used as an extra abstraction layer on top of the original data: one sometimes consider views as belonging to an *external layer* that users access, on top of the *logical layer* of relational tables that organize the data, on top of the *physical layer* of indexes and data storage structures. Separating views from data allows them to be used for confidentiality purposes: the view *PredictedCredits* can be published and used for statistics purposes, without the identity of the students being unveiled. Virtual views are also crucial in data integration contexts, when a source does not provide access to all its data, but just to a view over its data.

Materialized views are generally used for optimization purposes: if a complicated query or sub-query is often used, a materialized view defined by this query avoids repeating the same computation again and again, functioning as a cache over the data. This can be done in two ways: either the user needs to refer to the materialized view in the query for it to be used, or the materialized view is automatically used whenever useful, the database engine rewriting the query to make use of this cached data, typically using query containment tests to check for the usability of the view.

All relational DBMSs support virtual views. Materialized views are not part of the SQL standard (there is the possibility of defining tables by a query, but such tables are not *maintained* as we discuss further) but major systems offer the possibility of creating them, with a proprietary syntax. At creation, it is generally possible to specify whether the materialized view should be used for optimization purposes when finding a rewriting of a query (e.g., the `ENABLE QUERY OPTIMIZATION` clause of DB2's materialized query tables).

Updates in Relational Databases

Most research about views in relational databases relates to their interaction with database updates. Until now, we have mostly had a static view of databases: the content of tables is fixed, and we interact with them through queries. Obviously, in most applications, tables change over time, as new data appears, data gets modified, and old data is removed from the database. These three basic operations can be carried out in SQL as follows:

```
INSERT INTO Students VALUES(5,'Alice','Liddell', 0)
UPDATE Students SET credits = credits + 10 WHERE id = 3
DELETE FROM Students WHERE credits < 50
```

These three update operations respectively insert a new student in the database, increase the credits of a given student, and delete a student from the *Students* table. Note that the location of the tuple to update or delete is given in the `WHERE` clause similarly as it would be expressed in a query: the idea of using locator queries to express updates is a very general one.

Two fundamental problems arise when views are defined over dynamic data. First, materialized views need to be updated whenever the result of their defining query changes because of updates in the database; this is known as *view maintenance*. Second, since views can be used as an external layer that users can query without knowledge of the logical organization of the data, they should also be able to update the data through views, and this update should be propagated to the original data: this is the *view update* problem. We now elaborate on these two problems.

View Maintenance

The view maintenance problem is to determine how to efficiently maintain an up-to-date materialized view when its base tables are updated. Consider again the view *PredictedCredits* that we assume has been materialized. It is clear that each time a student is removed from the *Students* table, or the current credits of a student are updated, or a new tuple is inserted into the *Enrollment* table, the relevant portion of the view needs to be updated as well.

In order to avoid unneeded computations, the system needs to detect whether a view can be affected by a given update, i.e., whether the update is *relevant* to the query. For instance, no update in the *Dependencies* table, and no modification of the name of a student, can have an impact on the *PredictedCredits* view. Static analysis approaches can be used to determine the potential impact of an update on a view, independently of the current data. Once an update is found relevant to the view, the query defining the view can be evaluated again and the view reconstructed.

In most cases, it is possible to do better, with an *incremental maintenance* approach, that aims at avoiding this recalculation step together, and just incrementally maintaining the view by adding or removing individual tuples. Let us see a practical example, with the simple yet elegant *counting algorithm* (Gupta et al. 1993) for incremental view maintenance. Consider the following (materialized) view, that lists all courses at least one student is enrolled in:

```
CREATE VIEW Courses AS
  SELECT DISTINCT e.course FROM Enrollment e
```

The main idea of the algorithm is to store in the view, in addition to the tuples, an extra counter that indicates how many *derivations* of this tuple can be found in the database. For example, the "databases" course appears twice in the table *Enrollment* so there are two different derivations of the tuple ("databases") in the view *Courses*. Consider now an update on the table *Enrollment*. For simplicity, we assume it is either an insertion or deletion, modifications being dealt with as a sequence of a deletion and an insertion (it is possible to extend the algorithm to deal with modifications in a direct manner). We describe how the view is maintained. If we deal with an insertion, let c be the projection of the new tuple on the attribute *course*. The value c is searched in the materialized view *Courses*. If it occurs, the corresponding count is incremented by 1; otherwise, it is inserted, with a count of 1. For a deletion, we proceed similarly: we decrease the counter associated with the course deleted, and if it reaches 0, we delete the tuple from the view. Such a simple procedure can be defined for a large class of queries, with support for aggregation or negation. For a broader outlook on view maintenance approaches, see Gupta and Mumick (1995).

DBMSs that support materialized views, such as Oracle or DB2, allow specifying at view creation time whether a view should be maintained automatically, and, if so, whether the view should be entirely recomputed after each update operation or incrementally maintained (when possible) (Lorentz 2010).

View Update

The view update problem is the converse of the view maintenance problem. Instead of determining the consequences on a view of an update on the database, we now look at how to translate on the databases an update on a view. Imagine a secretary with access to the sole view *Courses* needs to change the name of the "automata" course to "formal languages". Does this make sense? In other words, is there any reasonable translation of this update operation on the table *Enrollment*? In this case, it seems there is: just replace every occurrence of "automata" with "formal languages" in the table *Enrollment*. What if someone with access to the *PredictedCredits* view wants to change the credits of a given student? Now, there does not seem to be any reasonable way to translate this into a database update operation, since the *credits* attribute of the view has been computed as an aggregate of several values, and it is unclear which value should be changed.

The view update problem consists in determining in which cases updating a database through a view makes sense, and when it does, what the most reasonable translation of the view update is. In Keller (1985), algorithms are proposed for view update translation when views are defined using simple conjunctive queries with all join variables exported in the view. In addition to these algorithms, this work is of particular interest because it identifies a number of criteria that a view update translation should verify to be "reasonable":

1. The translation should not have any effect on the tuples not exported in the view.
2. The translation should affect at most once a database tuple.
3. The translation should be minimal, i.e., there should not be unnecessary operations.

The SQL standard supports updatable views only when the view is defined using a single table, and no aggregation or grouping is used. Implementations may go beyond that and sometimes allow updating simple multiple-table views. The standard `WITH CHECK OPTION` clause that can be used in view definitions states that updates that would cause changes to tuples not visible in the view should be disallowed.

Plan Generation

Looking back at Fig. 10.2, we arrive at the step where query plans are generated and the physical plan that will be run on the actual data is chosen. In order to decide on a query evaluation strategy, query optimizers are built out of three components:

- Logical rewriting rules and index access strategies that are used to generate, given a query, its possible execution plans

- A cost model for estimating the cost of a plan, typically based on the expected number of disk accesses and CPU use of every atomic operation; for the estimate to be precise, it needs to be based on statistical information about the data
- A search strategy that guides the optimizer in exploring the space of possible evaluation plans

The design of the query optimizer, and in particular, the heuristics for estimating plan costs, are an important component of database management systems, that is kept as a secret in commercial DBMSs. We now present in more detail research about using histograms for storing statistical information, and how these statistics can be used to estimate the cost of a query, before presenting the architecture of a typical query optimizer. For further reading on query optimization, we refer the reader to (Chaudhuri 1998).

Cost Estimation and Histograms

Consider the query $Q_{>50} = \sigma_{credits>50}(Students)$. Such a simple query has usually at most two possible evaluation plans: either the table *Students* is linearly scanned, and all tuples with more than 50 credits are returned, or an ordered indexed on the attribute *credits* is browsed to retrieve all relevant tuples. The *Students* table is probably stored in the order of its primary key, *id*, however; this means the index of *credits* is a *secondary* index that stores, for each possible value, a list of pointers to all corresponding tuples in the database (a *primary* index on *id* could avoid this extra indirection).

Let us try to build a cost estimate of these two query plans, based on a simple cost model that only looks at the number of disk *pages* accessed. A page is the elementary unit of storage used by the DBMS; retrieving the whole content of a page is considered as an atomic operation, while accessing another page requires a costly random-access seek. Assuming a typical page size of 4 kilobytes and that 64 bytes are required to store each tuple of the *Students* table, the whole table uses $N/64$ pages where N is the number of students. Consequently, a linear scan of the table has a cost of $N \times 64/4096 = N/64$. The cost of using the index can be decomposed as follows: first, looking up 50 in the index; second, accessing all index entries for value equal or greater than 50; third, accessing all tuples pointed to by these index entries. Let C be the number of different credit values. Assuming storing a credit value requires 2 bytes, index lookup using a B^+ tree structure has a cost of $\log_{512}C$ (512 is half the number of entries one could store in a leaf node, see Silberschatz et al. 2010). The number of pages in the index entries accessed is roughly $N_{>50}/1024$ if 4 bytes are used for storing a pointer. Here $N_{>50}$ is the number of entries having value above 50. Finally, the number of pages accessed while retrieving tuples can be as large as $N_{>50}$ since two successive tuples are typically not contiguous (it would be possible to refine a little bit this estimate by considering the probability that a tuple is in the same page as a previously accessed tuple, provided that this page was cached). Summing up, using the index is cheaper if:

$$\frac{N}{64} \geq \log_{512}C + N_{>50}\left(1 + \frac{1}{1024}\right)$$

In most practical situations, the first and last terms of the right-hand side are negligible, and the question becomes whether $N_{>50}$ is less than or equal to $N/64$. To decide, we need statistical information about the credit values, usually stored in a *histogram*.

A histogram is a summary of the distribution of the values of a relation attribute, formed of a fixed number K of buckets, chosen small enough so that this summary can be stored in main memory and used by the query optimizer without incurring the cost of a disk seek. For each $1 \leq i \leq K$, bucket i contains statistical information about tuples for which the attribute value is between v_i and v_{i+1}. Thus, v_1 is the minimum attribute value and v_{n+1} the maximum value. The information stored is typically the number of distinct values in the interval $[v_i; v_i + 1)$, the number of tuples containing a value in this interval, and possibly other statistics of interest, such as the mean or median value in each bucket. A histogram can be used to estimate the number of results to a range query such as $Q_{>50}$: add up the number of tuples in buckets whose range intersects the range of the query, possibly refining the estimate for buckets that are at the boundary.

There are different ways to organize attribute values into histograms. *Equi-width* histograms partition the set of values $[v_1; v_{n+1}]$ into K intervals of the same size. This scheme is well adapted when the data distribution is close to a uniform one, but fails when the distribution is too biased. Going back to our example, assume that the minimum and maximum number of credits are respectively 0 and 1,000, but most students have credits less than 150. If we construct an equi-width histogram with 5 buckets, most of the data values are represented by the bucket $[0; 200)$, and the histogram is not very helpful to estimate $N_{>50}$. To avoid this issue, it is possible to use *equi-height* histograms, where the buckets are constructed such that the number of tuples per bucket is more or less uniform. In our example, this means that several buckets cover the interval $[0; 150]$. An estimate of $N_{>50}$ adds up the total number of tuples in all buckets whose lower bound is greater than 50, plus a fraction of the number of tuples of the bucket where 50 is contained, which yields a more precise approximation. An equi-height histogram, however, is more difficult to maintain in the presence of update operations than an equi-width histogram.

Research on histograms has aimed at proposing new ways of splitting the data values into buckets (e.g., v-optimal histograms (Ioannidis and Poosala 1995) whose frequency estimates are provably optimal for a large class of queries), at maintaining histograms when the data is updated (analogous to the problem of view maintenance), at efficiently computing histograms from the base data (mostly with the help of sampling techniques), or at building join summaries for the distribution of several attribute values, to deal with queries with multiple selection criteria. We refer to Poosala et al. (1996) for more details. DBMSs typically use both equi-width and equi-height histograms (Breitling 2005) and allow choosing between the two when tuning a database.

The Cascades Optimizer

We now explain briefly how a real query optimizer might generate a number of possible query plans, using logical optimization rules and available data access methods, evaluate their cost and decide on the plan to run. One of the main problems is to avoid a combinatorial explosion that would result in trying to apply all possible transformations. We take the example of the Cascades query optimization framework (Graefe 1995) that was intended as the basis of the query optimizer of Microsoft SQL Server. All logical optimization rules (*transformation rules*) and data access methods (*implementation rules*) are described as algebraic rewritings of a query plan. Each query plan is associated with its cost, which can be computed from the costs of the sub-plans. Cascades optimizes a query in a top-down manner using *memoization* to remember the optimization decisions for each encountered sub-query plan. When optimizing an expression, the system first considers if this expression (or one that is "similar enough") has not already been optimized, and, if so, directly uses the result. Otherwise, transformation rules and implementation rules applicable at the top-level are applied, using the guidance of the predicted cost of the resulting query plan, They also use heuristics that bias the exploration strategy, and *promises* for each rule that can be used to condition its application depending on previous and subsequent rule applications, in a goal-driven manner. Sub-expressions of the query are then optimized one by one, following the top-down process.

Data Indexing and Storage

Once a query plan has been selected by the optimizer (see Fig. 10.2), it is executed, using the indexes and data storage structures referred to in the plan (remember that the different methods of accessing the data have been considered and their cost estimated when optimizing the query). Most DBMSs index and store the data in a similar way: ordered datatypes are indexed using B-trees or B^{+}-trees (Bayer and McCreight 1972; Comer 1979), unordered datatypes with hash tables (sometimes dynamically maintained (Fagin et al. 1979; Litwin 1980)), and whole tuples are stored either in the nodes of the B-tree or B^{+}-tree index for their primary key, or sequentially sorted along their primary key, aligned with disk pages. A large body of research was dedicated to improve and build variants of these classical data structures, widely used for generic database applications. We present now research on alternative indexing and storing strategies that have been widely used for specific kinds of data: multidimensional indexes to efficiently retrieve objects based on their locations in Euclidean spaces, column stores that organize the data column-by-column instead of the traditional row-by-row storage strategy, and stream databases where data is not stored at all but queries are processed continuously as data arrive.

Multidimensional Indexes

B-trees and their variants are used for indexing linearly ordered data (integers, character strings, etc.) and efficient processing of point or range queries over the indexed attribute, i.e., selections like $\sigma_{credits=50}$ and $\sigma_{credits>50}$. Imagine now that the *Students* table contains two additional columns, *lat* and *long* that respectively contain the latitude and longitude of their home address, and that we want to retrieve the list of students who live less than 10 km away from campus. We could index these two attributes with a B-tree, but B-trees are not well adapted to this kind of query, and the best we could do would be something like computing the minimum and maximum latitudes covered by the 10 km radius, retrieving all students with latitude in this range, and then checking for each of them whether their combined latitude and longitude fall in the 10 km radius. Indexing structures have been proposed to better deal with such queries, such as quadtrees or R-trees.

Let us start with quadtrees (Finkel and Bentley,1974). Assume that latitudes and longitudes of student home addresses form a multiset of two-dimensional points. A quadtree divides a bounded region of 2D space (say, a square containing all points) into subregions in the following way. The square is divided into four squares of equal size, and each subsquare is divided again, recursively. A square is not divided further when it contains less than K points, for a fixed threshold K. This division of the 2D space naturally defines a tree of arity 4: the root of the tree is the whole region, the children of a node are the four subsquares of this node, and leaves point to the $\leq K$ points contained in the corresponding region. The construction and maintenance of such a structure is relatively easy. To answer the 10 km radius query, we retrieve, by a top-down browsing of the quadtrees all leaves that intersect the 10 km disc. Points in leaf regions entirely contained in the disc are returned immediately, while points in leaf regions that only partially intersect the disc are filtered one-by-one.

Quadtrees usually provide an efficient way of answering a geographical query, but they have one weakness: if the distribution of points is too biased (e.g., if many students live on campus accommodation, very close to each other), the tree may become quite unbalanced. Furthermore, in contrast to B-trees, the arity and depth of the tree are not optimized with respect to the number of disk pages accessed while searching the trees. R-trees (Guttman 1984) provide a solution to both of these problems. Again, the space is divided into a number of rectangular regions that are organized in a tree, but now the regions may overlap, are of arbitrary size and shape (though the region corresponding to the parent of a node is still a proper superset of the region of this node), and the tree is organized like a B-tree, with a large arity that is computed so that each tree node fits into a single disk page. Algorithms for searching and updating the R-tree are move involved than for quadtrees and directly inspired by their counterparts in B-trees. Again, answering the 10 km radius query means retrieving all leaves that intersect the 10 km disc, and subsequent filtering of the points contained in the leaves. R-trees generalize more easily to arbitrary dimensions than quadtrees, for which the arity is necessarily an exponent of the dimension.

Quadtrees and R-trees are widely used in geospatial extensions of DBMSs (Kanth et al. 2002), to index multidimensional data. They illustrate how the database community has proposed efficient data structures when new datatypes and applications of database technology appeared.

Column Stores

The main idea of column stores (Stonebraker et al. 2005) is simple: instead of storing tables row-by-row, with a whole tuple stored contiguously, they store tables columns-by-columns. Assuming again a page size of 4 kilobytes and 64 bytes for storing a tuple of the table *Students*, a traditional DBMS stores 64 tuples per page. Assuming 2 bytes for the *credits* attribute, a column store puts 2,048 values of this attribute in a single page. The interest of column stores is immediate in this numeric example: computing an aggregate of the *credits* attribute across all students, such as its average or sum, requires 32 times less disk page accesses than with a row store. Not all operations benefit of this data storage organization, however: any operation that needs access to all tuple values, such as computing a full join between two tables, or inserting individual tuples, typically requires more random seeks in a column store than in a row store. Generally speaking, applications that heavily use aggregates, statistics computation, or more generally individual attribute values rather than whole tuples, usually benefit from column stores. These applications are sometimes called *online analytical processing* (OLAP), in contrast with *online transaction processing* (OLTP) that cover more traditional database applications, such as order processing or banking. Another advantage of column stores is their ability to use more effective compression mechanisms to reduce the size of the data store, since data of a single type are stored contiguously.

Commercial (e.g., Sybase IQ, Vertica, KDB) and open-source (e.g., MonetDB) column-oriented databases coexist with traditional DBMSs. For a long time, all database-related tasks had been handled by traditional engines; the emergence of column stores might be an illustration that there is room for technologies that depend on the applications (Stonebraker 2008).

Stream Databases

Following up on the idea that classical DBMSs may not be adapted to all database management problems, we now consider the case when data is produced in such a large volume or at such a high rate that it cannot even be stored. A typical example is network data (see Chap. 7.): IP packets that go through a router of the Internet core are too numerous to be stored on disk. If one needs to query these packets (selection, aggregation, grouping), e.g., to detect potential attacks or trends in the use of the network, one needs to reverse the model: instead of evaluating various queries over a somewhat fixed collection of data, we want to evaluate a fixed set of queries over continuously streaming data. This is the model of stream database systems, such as Gigascope (Cranor et al. 2003).

For flexibility, one would like to use a general-purpose query language like SQL to query the stream of data. Note, however, that since it is impossible to store the entirety of the stream, some queries cannot be evaluated, such as those involving arbitrary joins with past or future data. For this reason, Gigascope defines a restriction of SQL where queries need to be evaluable in a *sliding window* of fixed size. All relevant packets in this sliding window are typically stored in memory. Query optimization has very different constraints than in classical settings to avoid missing some of the packets, it is critical to reduce the amount of data kept in memory by pushing the operations with the highest selectivity as early as possible, sometimes even implementing them in the code of the network interface controller. Higher-level operations (joins, grouping, etc.) can be applied later on the buffered data. It is also possible to increase performance by partitioning the stream and have each substream handled by a different computer; this partitioning, however, must not put in two different groups packets that need to be used together to answer a given query, which implies basing the partitioning operation on the query (Johnson et al. 2008).

A number of prototypes and commercial systems for database stream management have appeared. Even more so than for column stores, their applicability is restricted to very particular scenarios: network traffic, real-time auction market analysis, etc.

Hardware and Why it Matters

We are now at the bottom of the query processing system, where the query is executed on actual hardware. Perhaps even more so than for other software, the performance of database management systems has been strongly tied to the evolution of hardware architectures. The design of cost models, index structures, storage engines in traditional DBMSs has been based on a number of assumptions on how hardware functions:

1. High cost of disk accesses, and, especially, random seeks
2. Limited amount of available main-memory
3. Mostly serial CPU instruction processing model
4. Relatively low network bandwidth

However, as elaborated in Chap. 5, hardware and network infrastructure have evolved to the point where the validity of all these assumptions can be questioned:

1. The recent advent of flash memory and solid-state disks radically change the performance of disk accesses (see below)

2. The amount of main memory available in even low-end PCs makes it possible to store database indexes and sometimes even the data itself in main memory (Garcia-Molina and Salem 1992)
3. Parallel architectures are more and more frequent, to the point where standard modern graphics processor units are able to process hundreds of parallel execution flows, which makes them suitable for some database management tasks (Govindaraju et al. 2006)
4. The network bandwidth in a local cluster can be higher than disk transfer speeds, which has the ability of making main-memory distributed DBMSs more efficient than centralized disk-based ones (Apers et al. 1992) (see the next section for a discussion of distributed databases)

These examples explain why the evolution of hardware architectures does matter for database management systems, and why research on understanding and exploiting new capabilities of hardware is an active component of database research. We illustrate with the example of solid-state drive for database storage.

SSDs vs Magnetic Hard Drives

Secondary (i.e., non main-memory) storage of data in DBMSs has mostly relied on magnetic hard disk drives. These disks are made of one or several rotating platters where information is encoded by the orientation of the magnetic field generated by localized regions, organized in concentric *cylinders* and radial *sectors*. Information is read and written with magnetic heads that hover over the platter. Reading or writing to an arbitrary region of the disk requires a *random seek*: the head of the appropriate platter needs to be positioned over the correct cylinder and then wait for the moving disk to reach the correct sector. A *sequential read* that retrieves contiguous portions of data, on the other hand, is much faster since the head can remain fixed and the data is read as the disk rotates. The order of magnitude of the seek time and read sequential data transfer rate are, for a modern disk, respectively 10 ms and 50 megabytes per second.

Since the mid-1990s, a new form of permanent data storage has appeared: *flash memory*, using *floating-gate* transistors, transistors wired on chips so as to store an amount of charge for extended periods of time. Recently, the technological advances in building flash memories have led to the commercialization of solid-state drives (SSDs for short), which are drop-in replacements for hard disk drives formed of an array of flash memory units. The absence of any mechanical parts in such drives leads to negligible seek times. Modern SSDs have read data transfer rates comparable to that of magnetic drives, whereas sequential write transfers are somewhat slower (but random writes are typically faster).

The near-absence of seek times makes SSDs particularly suitable for database applications, where it is common to read small data blocks scattered across the storage area. Recent studies (Lee and Moon 2007) show that, indeed, read performance of DBMSs can be dramatically improved by using SSDs.

However, using a traditional DBMS on a solid-state drive causes other forms of problems, related to an inconvenience of flash memories: it is impossible to update a data item in place without first erasing the corresponding block of flash memory. This means that actual writing speeds are considerably slower than expected. To avoid this issue, Lee and Moon (2007) proposes to implement updates by logging all update operations in a fragment of each memory block kept free for this purpose. Reading the current state of the data consists thus in reading the base data, and updating it in memory with the extra logged updates. When the logging area is full, the whole block is erased and rewritten. This approach, which heavily relies on the behavior of flash memory, in the same way as traditional indexing approaches heavily rely on the fact that disk seeks are costly, allows obtaining improved performance over a regular DBMS on magnetic drive, even when considering update operations.

Another important aspect of building database systems with SSD storage is to understand the precise behavior of SSDs. A number of SSDs are thus benchmarked in Bouganim et al. (2009), exhibiting some counterintuitive results. For instance, despite the lack of mechanical parts, some seek latency appears, mostly because of the overhead introduced by controlling software. Another observation is that SSDs often do not exploit the possibility of parallelizing reads and writes operations over the flash memory arrays. Some of these characteristics are likely to be transitory behavior of SSD controllers, while some others will be important in designing future database management systems.

Distributed Databases

Before concluding this section on core database research, we want to mention the important aspect of distribution in database management systems that pervades the entire query processing pipeline. We say that a database system is *distributed* when the data itself is spread over a number of computers (also called peers, or hosts) connected over a network (See Chaps. 7, 8, 9.) The role of a *distributed DBMS* is then to manage this distributed database and to “make the distribution transparent to the users” (Özsu and Valduriez 2011). There are several reasons why we might want to distribute data:

- Data may be distributed to begin with, because of organizational reasons. Think, for instance, of the human resource and sales data of a company, which might reside in different departments, possibly in different physical locations, but sometimes needs to be seen as parts of a single database, e.g., for business intelligence purposes. At the extreme, the World Wide Web may be seen as a gigantic database consisting of data distributed all over the planet, seen as a whole by applications such as search engines.
- Data may simply not fit on the disk(s) of a single computer, however large they may be. A database that records stock market transactions, or meteorological

data, for instance, may get to enormous sizes: several hundred of terabytes for the database maintained by the Max-Planck-Institute for Meteorology (WinterCorp 2005).

- Non-distributed databases provide a single-point of failure: should the computer hosting the database fail, or should the number of data access exceeds what the database management system is capable of handling access to the entire database would be lost. Conversely, if the data is distributed, the load is divided between all peers, and a failure of a single host only affects part of the data. Availability can even be guaranteed to some extent if data is *replicated* over several peers of the network.
- It is possible in some cases to distribute data to improve the efficiency of query evaluation. We have already mentioned that a distributed database with data in main memory may be more efficient than a traditional local database with data on disk. Even when data is stored on disk, distribution allows parallel processing of a query. Recall that queries of first-order SQL can be evaluated in constant time on a parallel machine, which means that first-order SQL query evaluation can be very efficiently run in a parallel manner.

How data is distributed over the network depends on the reason data is distributed. When distribution is inherent in the organization of the database, there is no choice and it is often the case that data is stored in a heterogeneous manner and needs to be *integrated*, as we explain further in this chapter. When data is distributed for size, reliability, or optimization reasons, different data distribution strategies can be selected. Most commonly, relations are either *horizontally* or *vertically fragmented* (Özsu and Valduriez 2011). In horizontal fragmentation, a table is partitioned along its tuples, and groups of tuples are stored in different peers of the network. In vertical fragmentation, the partition is made according to the attributes of tuples, and each peer stores a subset of the attributes of each tuple, as well as its primary key. This storage choice is reminiscent of the distinction between row stores and column stores, and similar tradeoffs arise. Another point of interest is the network architecture used, which can range from centralized settings where a master host, connected to a number of slave hosts, acts as an entry point to the database, to distributed tree structures or fully distributed models such as distributed hash tables over peer-to-peer networks (Abiteboul et al. 2011).

A number of traditional database problems, such as query optimization or transaction management raise radically different challenges in a distributed environment. In some cases, this has led to relaxing some of the constraints traditionally imposed by relational DBMSs. This trend, sometimes dubbed the *NoSQL* movement, has resulted into distributed data and computation systems that do not support ACID transactions and have limited expressive power, but very high efficiency on extremely large collection of data, such as the MapReduce framework (Dean and Ghemawat 2008), extensively used by companies such as Google to process petabytes of data a day.

For an in-depth discussion of distributed database systems, we refer to the textbook (Özsu and Valduriez 2011).

Research on core database technology covers a large spectrum of areas, from logics to systems and optimization issues, even up to the benchmarking of modern hardware. We now move to a different vein of research, to extend the main approaches that have made the success of relational databases (abstraction, algebraic representations, etc.) to cover other applications and functionalities.

Extending Database Functionality

We have stressed that the relational model is a natural evolution point as data management systems increase in their abstraction – hiding from the programmer or end-user details of the physical layout of data and the implementation of queries. But in some ways the relational model is low-level: the data model (at least, as visible to the data definition language) imposes quite a few restrictions, including allowing only a fixed set of simple data types as attributes of a tuple, requiring the data developer to spend time "breaking down" information into small components. Indeed, this is a basic part of the philosophy of the relational paradigm towards data design. In addition, the set of features in a relational schema – particularly with regard to integrity constraints – are very limited compared to the kinds of semantic restrictions that one may want to express about real-world data.

Much of the research in the database community has revolved around extending the mathematical foundation of database systems to be less "low-level" (in data model). Another direction has been to look at richer data definition languages, even within relational databases. A closely-connected topic is the ways of building up larger datasets from components – data integration. Some of these extensions have been pursued while trying to preserve the relational approach in its entirety. For example, in the case of query languages for XML documents, database research still takes a declarative approach, compiles into an algebra, and applies rule-based optimization. In other cases essential features of the relational paradigm are jettisoned completely. We will take a quick look at each of these general lines of research within this section.

Data Design

The relational database model is built on a very simple data structure, a table where each cell contains a simple type. Nevertheless, it was seen that one can represent the information in many applications using relations, by breaking down more complex structures into tables. But exactly how should complex data be translated into tables?

The major challenge is that there are many ways of representing the same information. In the university example, we had a *Students* table including *id*, *first*, *last*, and *credits* (the student's current credits), and an *Enrollment* table that included *id*, *course*, and *credits* (the course credit value for this student). But one could also have one large table *StudEnroll* that had all of the previous attributes. The second possibility seems odder, but can we say that it is worse?

The subject of *data design* originated very early in database research; its goals included:

- Capturing a notion of two schemas representing the "same information"
- Formalizing the notion that one schema is better than another
- Providing algorithmic techniques for getting to a good schema

These goals could be seen from the same two-pronged perspective that defines database research as a whole. On the one hand, in attempting to define the notion of "information equivalence", database research was exploring the "theory of information" in a very grand sense. Certainly an insight into what constitutes the same information content within data would be significant even if it was not accompanied by effective methods. On the other hand, data design research had a pragmatic goal of offering advice to database designers on how to create and maintain database schemas.

It should be clear that such a project must limit its scope in some way. First of all, there are many human factors in determining what a good schema is – database research cannot say if one column name is better than another, or whether it is better to store the yearly salary or the monthly salary of employees (since clearly one can derive one from another). Thus the best we can hope for is that computer science research could identify certain designs as being inferior or superior to others: we cannot hope to identify a unique "best design". Second, such a process must take as input some information about the semantics of the database, not just that which is captured in standard table meta-data. For example, if we only know that we have a table named *StudEnroll*, including columns *id*, *first*, *last*, *course*, *creditsObtained*, and *creditsCourse*, but with nothing about its meaning, we cannot identify that there is any shortcoming. Data design thus starts with a description of the "semantics of information to be stored" in some richer data model (these may include nested tables, lists, sequences, or other higher-level structures), and then gives a method for translating to a relational database schema. *Entity-relationship* diagrams represent one such formalism for high-level data description; there is a simple algorithm for generating a relational schema from an entity-relationship diagram. More powerful modeling languages, such as UML, can also serve as a starting point.

The most well-developed theory of "better design" has looked at simpler languages for describing the semantics of information. Most of the algorithmic results work in a simple modification of relational data definition languages, in which information is described using a set of tables plus integrity constraints. These include SQL key and foreign key constraints, as well as more powerful constraints. The paradigmatic example uses *functional dependencies* as the constraint language. A functional dependency states that a subset of the columns determine other attributes of the table. In the *StudEnroll* example, we have that the value of *id* determines the values of *first* and *last*.

The standard theory contributes a notion of "better schema", formalizations of "information equivalence", and a method for going from bad schema to a better one.

The problem with the schema above can be identified formally by the presence of a functional dependency (*id* implies *last*) that does not follow from a key constraint (*id* is not a key, it is repeated in multiple rows): such a dependency implies that information in some columns in the row is redundant, and hence will be repeated many times. If physical storage reflects the repetition in the table structure, then this will clearly lead to performance issues, as well as extra infrastructure needed to maintain consistency during updates. The difficulty is summarized as: a piece of information should only be represented in one place, and to change it one should only need to modify in one place. A schema that includes functional dependencies is said to be in *Boyce–Codd Normal Form* (BCNF) if all functional dependencies follow from key dependencies.

What does it mean for two schemas to have the "same information"? One well-studied definition is that a schema B is a *lossless-join decomposition* of schema A if tables in A can be obtained by joining projections of tables in B. The schema consisting of tables *Students* and *Enrollment*, with the obvious key dependencies, is a lossless-join decomposition of the *StudEnroll* table; the lossless-join property states that *StudEnroll* can be exactly recaptured using the join $\mathit{Students}\bowtie\mathit{Enrollment}$.

Algorithms exist (Codd 1975) for automatically finding a lossless decomposition of an arbitrary schema into a BCNF schema. Normalization can be seen as a design methodology; start with an initial design – for example, one reflecting the user interfaces that end-users would like to see. Then continue to decompose until a normal form schema is obtained. The original tables can be re-captured either as ad-hoc queries, or as materialized or virtual views. If the un-normalized tables are materialized, many of the space benefits of normalization are lost. But even then the benefits for software infrastructure will remain; updates will need to be specified only on one table, and any update of redundantly-stored information will be done automatically.

From a theoretical point of view BCNF decomposition is the most basic example of *normalization theory*. Normalization has been considered for richer schema languages, including a number of other kinds of integrity constraints, such as multi-valued dependencies (Fagin 1977) and join dependencies (Fagin 1979). Stronger notions of information preservation have also been considered, such as being able to enforce all of the original integrity constraints on the decomposed schema using simple key constraints. In each case, the theory investigates whether or not equivalent schemas can be found for any schema in the data model. For example, a basic positive result in the theory is that for any schemas consisting of rich collections of integrity constraints (functional dependencies and multi-valued dependencies), one can find a lossless decomposition that requires only key constraints (Fagin 1977): the corresponding decomposition is said to be in Fourth Normal Form – a stronger normal form than BCNF. A sample negative result in the theory states that there is a schema S consisting of very simple integrity constraints, such that there is no lossless decomposition of S into schema S' in which key constraints on S' suffice to enforce all integrity constraints in the original schema. This result motivates enforcing weaker criteria on the decomposed schema (such as Third Normal Form (Zaniolo 1982)).

Normalization theory is an extreme example of the dual role of database research. On the one hand, a basic understanding of the virtues of normalized tables is considered essential for data designers and database consultants. On the other hand it represents a broad investigation into the meaning of information.

Advanced Data Definition

From Stored Data to Virtual Data

Data definition languages represent an important component of the relational model, playing a role analogous to type systems in general-purpose programming languages. The basic DDLs describe the attributes and attributes types in a set of tables and give integrity constraints that encode restrictions on the possible instances of the tables, along with relationships that must hold between tables. One kind of "relationship" is a foreign key constraint, mentioned already. Another extreme example of a relationship between tables is when one table is completely determined by another. This is exactly the case of *view definitions*, previously discussed.

Tables with foreign key constraints between them can still be updated independently, and a collection of such tables generally have the same "status" as a representation of the real-life facts to be stored. In contrast, the use of view definitions – whether materialized or virtual – requires a distinction in the kinds of tables that a database manager knows about, into those that are "basic" and those that represent derived data. Hierarchies of derived data can then be defined with views defined over views.

A particular use of virtual views is in *data integration*. Suppose we have several different database schemas $S_1 \dots S_n$ aiming at storing similar information. We wish to create a single unified interface to the data. Our first step is to come up with a single *global schema* S that can represent all information in any S_i. After that, we can give a logical definition of the global object in terms of the data I_i for each S_i stored on each local source. The single integrated database is a prime example of a *virtual database* – it can be given a precise definition, in terms of existing data, but it need not exist on any source.

How can we unambiguously define the integrated database? The simplest way is to create a query Q that takes instances $\vec{I} = I_1 \dots I_n$ over $S_1 \dots S_n$ and outputs a *global view instance* I over the global schema. The global instance I need not ever be materialized explicitly; instead the backend of the integrated interface generates queries to the appropriate S_i in response to queries over S.

The approach outlined above, often referred to as *global-as-view* (GAV), is conceptually straightforward, though performing the query-generation at runtime can be problematic. But the simplicity is misleading: when n is large a query Q

describing S in terms of the S_i may be difficult to write, and possibly impossible to evaluate. Furthermore, maintaining Q as the sources are modified is difficult, since its rewriting in terms of the source schemas may not even be human-readable.

More Complex Virtual Databases

An alternative is *implicit specification* of the global instance, as a data source I over the global schema S that satisfies various constraints with respect to the local sources $\vec{I}$. The most common approach to doing this, known as *local-as-view* (LAV) (Lenzerini 2002) describes I by giving constraints of the form $I_i \subseteq Q_i(I)$.

Given instances $I_i{:}i \leq n$ for the input schema, constraints of this form do not determine a unique database I, but rather a collection $Sol_V(\vec{I})$ of instances satisfying the constraints.

Although we cannot talk about "the integrated view", we can still make sense of querying an integrated view: the result of a query Q on the view is taken to mean the intersection of all $Q(D)$ for D in the collection $Sol_V(\vec{I})$. This set of results, often called the *certain answers of Q,* is equivalently seen as the set of facts of the form $t \in Q(I)$ that are logical consequences of the input data $\vec{I}$ and the view definitions relating $\vec{I}$ to an arbitrary solution I.

As an example, consider a data integration system that defines a virtual relation *Enrollment* with attributes *id* and *course*. One local source may have a stored relation *StudentIds* which has a single attribute *id*, while another might have a relation *Courses* with attribute *course*. The global view is related to the local sources by the mappings: $StudentIds = \pi_{id}(Enrollment)$ and $Courses = \pi_{course}(Enrollment)$. This defines the collection of enrollment tables that project onto the sources in the expected ways.

The advantage of the LAV approach in specification is fairly evident: specifications can now be much smaller, since they relate only two instances at a time. There is an enormous gain in modularity, since when a new source is added one must only write a new set of constraints involving only I and that source, and changes to the schema of a source I_i require only modifications to constraints involving I and I_i.

The disadvantage is also obvious: since the collection of instances satisfying the constraints is generally infinite, it is not clear how to calculate the tuples that lie in $\bigcap_{D \in Sol_V(\vec{I})} Q(D)$ at all, much less how to calculate them efficiently. A fundamental result is that for LAV views the certain answers for positive queries (an extension of the conjunctive queries) can be calculated efficiently in the size of the data (Levy et al. 1996). In fact, one can create a single view instance I that is "universal", in the sense that performing a conjunctive query on I gives the certain answers of Q with respect to the source instances $\vec{I}$ and the view definitions. One forms the universal instance I by simpling throwing in "dummy witnesses" that are implied by the view definitions.

For example, the view definition may state that for the integrated view I with attributes a, b, c, we have the requirement that $I_i \subseteq \pi_{a,b}(I)$ where I_i is a given source. Then for any tuple (a_0, b_0) in I_i, a solution I must have some value (a_0, b_0, c) in it. The universal solution is formed by choosing a distinct c for each such (a_0, b_0).

The ability to form universal instances gives an algorithm for determining the certain answers that is polynomial in the size of the data sources $\vec{I}$. This shows that implicit ways of defining virtual databases can still be efficiently implemented.

Integration vs. Extraction in Commercial Systems

The notion of data integration above is to create a virtual interface to data in different formats that may be accessed by external sources as if it were a centralized database. An alternative is to extract the data from the diverse sources and materialize the data.

One form of integration system, federated databases, has a fair amount of commercial support. For example, there is support in IBM DB2 and Microsoft SQLServer for creating views that refer to external databases. Federated database managers are not limited to relational databases (or to views defined via queries) but can encapsulate access to pre-relational or proprietary data, via hand-coded stored procedures.

Extraction-based approaches, in which explicit or implicit derived databases are materialized, are also supported by many commercial systems. IBM's DataStage product includes support for extraction based on implicit view definitions; most of the commercial usage of such systems, however, is done via manual coding of transformations.

More General Implicitly Specified Databases

There are many other formalisms that have been devised for defining virtual databases. Many extend the general contours of the LAV approach: the virtual database I is defined by a set of constraints that hold between it and stored data instances I_i. *Source-to-target dependencies* are one example of such constraints (Fagin et al. 2005). Care must be taken in both the constraints and the queries one is allowed to pose against the virtual database. Calculating the certain answers requires solving a satisfiability problem, and satisfiability is known to be undecidable for many query languages, including the relational calculus.

Another example of an implicit database formalism is that of deductive databases; in this case, a virtual database is defined by giving facts that it contains as well as axioms on the database itself, rather than on its relationship to other databases. Answering a query against the virtual database is again defined in terms of deduction: a fact is in the query result if the axioms and facts of the virtual database imply it. These axioms must be of restricted form in order for deduction to be decidable. For example, Horn clauses, of the form $\forall\vec{x}.R(\vec{x}) \rightarrow S(\vec{x})$, are one popular formalism for deductive databases.

Ontologies and Databases

Our last example of an implicitly specified database comes from *ontologies*. An ontology consists of a collection of facts coupled with axioms written in a restricted fragment of logic. The collection of facts is not taken to include all true statements about the real-world relation, but only a subset. The axioms are interpreted to hold not over the database of facts, but over the entire universe.

In the university example, we may have facts listing certain entities as being math professors, certain entities as being students, and some facts about which students are advised by which professors. For example, we may have a fact that student Bob Jones is advised by Rob Smith: *Advises*(Bob.Jones, Rob.Smith). We have an additional axiom stating that every math student is advised by a math professor. In the notation of description logics, this would be written: $MathStudent \subseteq \exists AdvisesMathProf$. This axiom is not treated as an integrity constraint on the set of facts: if it had been, then it would fail if Rob.Smith is not listed as a physics professor. Instead it is used to derive new facts. A query asking for all mathematics professors will then return Rob.Smith, since the axiom coupled with the fact base implies that Smith is a math professor.

As in the case of LAV integration, an ontology gives an incomplete description of a collection of data. Answering a query against an ontology is defined again in terms of certain answers: for a query Q, we return all the tuples t such that the database of facts and axioms derives that t is an answer to Q.

The exact formalism used for the axioms is restricted so that the derivation of facts can be effectively decided. A standard has emerged over a set of ontology languages, based on description logics (Baader et al. 2003) – a limited logical language in which all input relations must have at most two attributes. The queries must also be restricted, usually to be positive SQL queries without aggregation. Even with these restrictions, the complexity of the decision procedures is high: even for the simplest language, consistency of a fact is PSpace-hard (Schmidt-Schaubß and Smolka 1991) in the ontology. Nevertheless, the complexity for a fixed set of axioms, varying only the set of facts, is often manageable. Indeed, for many ontology languages, the certain answers can be determined using database methods: for any fixed query Q and ontology O based on axioms within the family, we can generate a first-order SQL query Q' that returns the derivable answers when evaluated on the facts. This is true of the commonly used DL-lite family (Calvanese et al. 2007), and also of more recent extensions (Calì et al. 2010) that subsume ontology languages and LAV-like data-exchange formalisms.

Ontology languages have been standardized by the World Wide Web consortium. The resulting family of languages, OWL (Horrocks et al. 2003) have a number of prototype implementations, in addition to limited commercial support. The approach via rewriting to a database language has been implemented in several research systems, particular QuoOnto (Acciarri et al. 2005). The main issue in these approaches is that ontologies may have thousands of axioms, and even an efficient translation may yield a query of size larger than current database managers can handle.

New Models: Complex Objects

Programming languages have available a rich collection of data types – they can form lists, associative arrays, vectors, and object classes that contain fields that may themselves be complex structures.The type system of relational databases in comparison is quite impoverished. Relational DBMSs manipulate "tables" or "relations" – from the theoretical point of view, a relation is a set of tuples, with each tuple being a function taking a column within a predefined set of column names to a datatype that is associated with the column. When we compare this to arbitrary programming language datatypes, we can see several dimensions in which they are limited: Relational tables are *homogeneous* – the data type within a table cannot vary row-by-row. They are also *flat*: although the columns of a relational database can have arbitrary built-in scalar datatypes, they cannot have any internal structure – or at least, no internal structure that can be referred to in the query language. Finally, they have no order and no duplication.

Of course, some of the use of complex data structures in programming languages is related to their more general mission – arrays and lists play a role in many fundamental algorithms, which are not intended to be implemented within a database manager. Still, much application data does have a rich internal structure, and relational databases often force users to use a structure that does not reflect their natural level of abstraction. This "impedance mismatch" has caused considerable concern in the database community, particularly since the move to relational database managers was prompted by a desire for greater abstraction.

We list these as limitations of the relational model, as espoused in papers and textbooks. Commercial database systems have worked from the beginning in a model that does not abide by these limits. They allow some limited heterogeneity by allowing certain cells of a row to be optional. The SQL query language supports this via primitives that can test for the presence or absence of a value for a given column. Although they generally require stored tables to be duplicate-free, they allow query results to contain duplicates, and SQL allows a query both to filter based on the position of a row in a result and to specify the ordering of the results. Nesting is supported as part of aggregate functions.

Still, the SQL extensions to support these are ad-hoc; the operators that support them cannot be freely composed, and some of them are available only at top-level. Researchers have tried to fill the gap between the theoretical model of pure tables and the SQL data model in practice, by developing a formal model that incorporates richer data types. The general goal is to follow the paradigm of the relational model: define a query language that (a) corresponds to a "natural" logic; (b) defines only polynomial-time queries; (c) can be translated into an algebra – defined loosely as a variable-free formalism.

Nested Structures

In the relational model, the basic object is a set of tuples. In the *complex-value data model* we can iterate tuple formation and relation formation, generalizing schemas to types that can combine aspects of both relations and tuples. Data types are build up from a given set of scalar types via tuple formation and table formation:

if $t_1 \dots t_n$ are types, and $a_1 \dots a_n$ are names, then there is a new type $\{\ a_1{:}t_1 \dots a_n{:}t_n\}$ whose instances are tuples, where a tuple consists of functions taking each a_i to an element of t_i;

if t is a type, then there is a new type $Set(t)$ whose elements are finite sets of objects of type t.

A database schema will then consist of a collection of objects of distinct types. Normal tables can be thought of as very special cases, of the type $Set(\{\ a_1{:}t_1 \dots a_n{:}t_n\})$, where t_i are scalar types.

Some special cases of the data model restrict the ways in which tuple formation and table formation can alternate – when these type-formers are required to alternate strictly (thus disallowing, e.g., a tuple whose attributes are tuples), the objects in this model are referred to as *nested relations*. At the query language level, the most well-studied proposal is *nested relational algebra* (NRA), defined initially for the nested relation model. NRA contains new operators for both navigating a complex-valued structure and building new structures. It includes the identity mapping on schemas as a basic query, and also the relational algebra operators product, renaming, and projection, extended to the nested case in the obvious way: for example, $\pi_{a_1 \dots a_m}$ is a query on objects of type $\{\ a_1{:}t_1 \dots a_n{:}t_n\}$ returning objects of type $\{\ a_1{:}t_1 \dots a_m{:}t_m\}$. Selection can be extended to nested relations by allowing selection conditions to include not only equalities of two top-level attributes but also identifications of scalar attributes nested within them.

The main new language feature is for nesting and un-nesting, The nesting operator is closely related to the GROUP BY construct of SQL. It is parameterized by a datatype t of the form $Set(\{\ z_1{:}t_1 \dots z_n{:}t_n\})$ where $z_1 \dots z_n$ are attributes, and a subset of the attributes $z_1 \dots z_k$. Given a set of tuples, it returns a set of nested tuples, where there is one nested tuple for every set of tuples that share the same values for $z_1 \dots z_k$, with each nested tuple having attributes containing the attributes $z_1 \dots z_k$ with their common values along with a new attribute containing the set of values for $z_{k+1} \dots z_n$ obtained for this group. Un-nesting acts as an inverse of the above operation, taking a set valued attribute and pairing it with all distinct values of the projection of an additional collection of attributes.

A related language with the same expressiveness is based not on nesting and un-nesting but on adding a mapping operation, which “applies a query pointwise”. One formalization of this is using an operator that takes as input a query $Q(x)$ taking objects of type t to objects of type t', along with a query Q' creating an object of type $Set(t)$; the result is a "set comprehension" $\{Q(x)|x \in Q'\}$ that applies Q to all elements of Q', producing a set of t' objects. A variant of this combines application with "flattening": the query Q being applied in this case produces $Set(t')$ objects, and the operator applies it to each element produced by Q', unioning the results to form an object of type $Set(t')$. Such a family of languages was defined in this way in Tannen et al. (1992), parameterized by a signature of basic operations, under the name *monad algebra*. In addition to the "flat mapping" operator, monad algebra has the basic operations of the λ-calculus (see Chap. 2, Appendix 2) along with operations for pairing and projection, union, and singleton-formation. Tannen et al. (1992) shows that when either a difference or an equality operator is taken as a primitive, the resulting language captures NRA.

How do the languages above fare in meeting the desiderata of query languages? There is strong evidence that the language is not "too expressive": queries are polynomial-time, and the Boolean queries expressed by the language are exactly those expressed in relational algebra – this is the *conservativity theorem* (Paredaens and Van Gucht 1992). An extension (Wong 1996) shows that it is never necessary to build up sets whose nesting depth is bigger than the combined depth of the input and output.

Is the language expressive enough? Evidence in the affirmative is that there are a number of languages that have equivalent expressiveness to it. In addition, if we look at natural representations of nested relations as flat relations, we find that not every relational query on representations is expressible in NRA.

Adding Support for Duplication

What about adding support for tables with duplicate rows, or sequences? In the absence of nesting, it is easy to simply re-interpret the relational algebra on bags; for example, the difference operator can be interpreted to subtract multiplicities of occurrences of tuples. It is more challenging to arrive at a query language handling nested bags, as is needed to model aggregate operators in SQL. The approach of the monad algebra can be easily modified to deal with a data model where the set type-former is replaced by a multiset or list type-former. The pointwise application operator is adapted to bags or list in the obvious way – a query is applied pointwise, preserving multiplicities in the bag case. For bags an extension of this type was formalized by Libkin and Wong (1997) and given the name Bag Query Language (BQL). Related languages are given in Grumbach and Milo (1993). BQL certainly satisfies some of the desiderata of the relational paradigm: queries are given algebraically (basically, by definition), and queries are in polynomial-time, in fact in LOGSpace (Grumbach et al. 1996)

What is the relationship of the BQL language to logic? One major difficulty encountered is in selection. In analogy with selection in relational algebra, we would like to be able to check if two bag- or list-valued attributes are the same. Set equality can be expressed as bi-containment, which requires checking membership of every element in one set in the other. For any fixed nesting, this check runs bottoms up with a check of membership on values, and hence can be expressed using a fixed alternation of the quantifiers ∃ and ∀ in first-order logic. In the case of bags, equality requires that the multiplicities of each element are the same, which requires some form of counting. In fact, in any of the basic bag query language, one can express basic cardinality constraints on flat structures – e.g., that the in-degree of a binary relation is the same as its out-degree – by using bag creation followed by a bag equality test.

So counting is somehow inherent to a bag query language. The expressiveness of BQL over flat bags can be characterized using an extension of relational algebra with arithmetic (Libkin and Wong 1993). Although this is arguably not a standard-enough logical language to say that BQL is "canonical", the characterization is useful for showing that certain queries cannot be expressed in BQL.

More Powerful Languages

The complex-value models above are analogs of the relational algebra and calculus, which capture restricted classes of the polynomial-time queries. But what are the analogs of relational languages with recursion? For example, in the relational setting, the language of *least fixed-point logic* captures exactly the polynomial-time queries, assuming the existence of an order. Abiteboul and Beeri (1995) defines an extension of the nested relational algebra with a powerset operator, and shows that it is sufficient to define recursive operators. Gyssens and Gucht (1992) and Suciu (1997) define more limited recursive extensions of nested relational algebra; Suciu shows that they have a conservativity property with respect to fixpoint logic over relations, while Gyssens et al. (2001) shows that the two languages have equivalent expressiveness.

We motivated the complex-value model via the impedance mismatch problem between databases and programming languages. But nested structures are only one feature of modern programming language type systems. One additional feature is that of pointer or reference types. Relational database attributes can model pointers, but not the ability to create new objects (or new references) dynamically in fixpoints. Abiteboul and Kanellakis (1998) takes the natural step of considering nested languages with both recursion and object creation. The resulting language is shown to be able to define arbitrarily complex database transformations. Seen from the limitative philosophy of the relational paradigm, this shows that recursion and pointer creation is simply too powerful a combination.

Data Design for Nested Structures

DBR has also been concerned with extending relational *data design* to complex objects. At the level of theoretical analysis, there has been considerable work on defining dependencies on nested structures and studying their interaction

(Hartmann et al. 2006). Normal forms have been defined, which represent schemas for nested objects with “less redundancy” (Ozsoyoglu and Yuan 1987; Mok et al. 1996; Tari et al. 1997; for a comparison, see Mok 2002).

Industrial Support and DBR for Complex Objects

In the 1990s, support for richer programming language types was seen as the natural evolution point for commercial database systems. It seemed clear that both the relational model and relational DBMSs would be supplanted by object-oriented counterparts. Vendors made two practical cases for objected-oriented systems. The first emphasized the software productivity increase obtained by diminishing the “impedance mismatch” between program objects and storage. The second was based on performance: by retrieving data object-at-a-time, rather than relation at a time, an object DBMS could improve performance in a way similar to the benefits of set-at-a-time over tuple-at-a-time.

But while object-oriented features are found in most database managers today, there is no convergence on an object-oriented paradigm for data management that replaces relational databases. DBMS software vendors have taken several distinct approaches to merging objects and databases.

- Some products add persistent storage to an object-oriented language, emphasizing integration with the type system of the language rather than faithfulness to the features of data management software. VOSS, for example, adds persistence and transaction support to the object-oriented language SmallTalk.
- Other products build a standalone object database management solution with several programming language interfaces, supporting both navigational access (following pointers) as well as an object query language. The language OQL (Cattell 1997) was proposed as a standard object query language for object databases (ODBMSs), corresponding to an extension of nested relational algebra with powerset of (Abiteboul and Beeri 1995) and introduced with the O_2 ODBMS (Deux 1990). The ODBMS Versant supports a variant of OQL, as well as a proprietary query language VQL.
- *Object-relational* database systems (ORDBMSs) build standalone database products on a more evolutionary approach, adding on object features to relational systems. A key for ORDBMS products is extensibility: they give the developer the ability to define new types and functions; some of them (notably PostgreSQL) give ways of extending the optimizer. The SQL-99 standard incorporates many features of object-relational systems.
- Object-relational mapping tools, such as Hibernate, provide support for storing programming language objects in relational tables. These tools work on top of a third-party relational database, providing languages for programmers to define mappings between objects and relations, and runtime APIs that implement

transformations on objects by translating them to calls to the DBMS. Some of these tools support their own object query languages.

Object-relational mapping tools are in widespread use, particularly for Web development; however, they do not replace relational DBMSs, but only supplement them. ORDBMs have had broader commercial success than ODBMSs: Postgres and its successor PostgresSQL are heavily-used open-source database products that embody many ORDBMs features. All major commercial vendors claim some support for ORDBMS features, but with wide variations: none of them support the full SQL-99 standard. Since the least common denominator capability over all of these systems consists of relational database management, it is difficult to say that the era of object systems has come.

The connection between the languages proposed by ORDBMS products and those proposed in DBR is radically weaker than for relational systems. The ORDBMS standard query language SQL-99 has little resemblance to the algebras proposed for complex objects. When we turn to object-oriented database design, we find that the gap between DBR and practice is even wider: not only are the normal forms for object-oriented databases not applied in designing ODBMS and ORDBMS schemas, the normal forms are barely known outside of the research community.

XML and Tree-Structured Data

XML data management arose as an application several years after query languages for complex values and objects appeared. In querying XML documents, one notices several features that were related to complex-value models. And it is in XML that the goal of extending the relational paradigm *en masse* to a richer set of datatypes has had the most commercial and theoretical success.

An XML document has many of the properties of a list-oriented model, an ordered variant of the bag models discussed earlier. In fact, documents could be coded as nested lists. Ignoring attributes for a moment, a document consisting of a root node with tag A and children $C_1 \dots C_n$ could be represented as a nested list whose first element is a singleton set representing A (e.g., using a one-attribute schema) and whose next n elements are representations of each C_i.

The close connection between documents and complex values gives a motivation for a query language built on an extension of the application operator (the "functional approach") mentioned above for complex objects. Consider queries Q that take as input a variable binding – a mapping of free variables of a query to a node in a document – and which output a *nodeset* sequence of nodes in some document obtained by enlarging the input document. Given Q and nodeset O, we iterate Q on O by applying it to all members of O in sequence, concatenating the resulting nodelists to get a new nodelist. This operator is the basis for the main operator of the XML query language XQuery. A *FLWR expression* in the XML language XQuery iterates a query *Q* over a list created by another expression *E*. For example, the XQuery query:

For $x *In* $*root*/*descendant*::*Prof*
  Return {*Prof* / *@lastname*}

returns the lastname attribute of every professor element in a document.

A major distinction between XML and list- and bag-oriented data models is that the latter have schemas that fix the nesting-depth of the data. Since queries in these former models have inputs that are typed to satisfy a given schema, this implies that any given query must only deal with structures of some fixed nesting depth. In contrast, an XML query should be able to deal with documents of arbitrary depth. The navigation or selection component of a query language must therefore have some mechanism for searching arbitrarily far down within a document. XML query languages borrow a mechanism from modal logic, the use of *path expressions* for navigation. A path expression consists of a command to move in a certain direction within a structure. In the case of XML documents, these directions are referred to as *axes*, and they are given relative to a node in a document. The *descendant* axis, for example, refers to navigation to any descendant of a node, while the *following-sibling* axis refers to navigation to a right-sibling. Path expressions are built by composing steps, which consist of axis plus tag-filters; an expression $x/*descendant*::A selects all descendants of the node (or nodes) associates with variable $x that are labeled with A.

The use of path expressions to navigate XML documents was pioneered by James Clark, who developed the language XPath, later standardized by the World Wide Web consortium (W3C 1999). XPath was not intended as a query language, but as a sublanguage for selecting nodes within an XML document; it was used within a variety of other XML languages, such as the transformation language XSLT. The original language was variable-free, consisting of compositions of navigation steps and filters, where filters could be built up from existence tests using built-in functions and operators. For example, *descendant*::*Prof*[not (*child*::*Advisee*)] is an XPath query that selects all professor element nodes that are descendants of a given node, where the node must not have any advisee-elements as children. Later versions of the language added variable bindings (W3C 2007a).

We have already mentioned the query language XQuery. XQuery was developed for "database-style" transformations on XML documents – data-intensive transformations that do not require a recursion procedure. XQuery combines the node selection facilities of XPath with the FLWR iteration facility – roughly speaking, modal logic combined with function application. Consider a document which contains professors (*Prof* elements) and their advisees (*Advisee* children of *Prof* elements), and which also redundantly contains students (*Student* elements).

The following XQuery query returns a list of professors who do not advise anyone, listing as children all the students without an advisor:

```
for $x in $root/descendant::Prof[not(child::Advisee)]
    return
    <Prof>{Prof/@lastname }</Prof>
    <CouldAdvise>
    for $y in $root/descendant::Student[not(child::Advisor)]
    return $y
    </CouldAdvise>
```

The outer *for* loop iterates the variable $x over the path expression returning the collection of *Prof* nodes without advisees; the inner *for* loop iterates variable $y over another path expression returning potential student advisees. The query also makes use of element constructors (such as <Prof>...</Prof>) that generate new nodes – these play a role roughly analogous to nesting in complex-valued models.

What can we say about XQuery in regard to the desired characteristics of query languages from the relational paradigm? The core of XQuery defines only queries with polynomial-time data complexity (Koch 2006). The variable-free language XPath itself satisfies even better bounds, having complexity polynomial in both the query and the document (Gottlob et al. 2002a); indeed, for a large fragment the combined complexity of evaluation is linear (Gottlob and Koch 2004). In terms of limitation on its expressiveness and relationship to logic, a conservativity theorem similar to that of bag query languages holds: the Boolean queries that are expressible in the XQuery core are exactly those expressible in first-order logic with an additional counting quantifier (Benedikt and Koch 2009). Furthermore, the fragment of XQuery Boolean queries that corresponds to first-order logic has been identified ("atomic XQuery" of (Benedikt and Koch 2009)).

At the level of industrial acceptance, XQuery has been quite successful, albeit not yet at the level of SQL. While languages for nested relations have been mainly confined to academia, XQuery has been standardized by the World Wide Web Consortium (W3C 2007b) as a Web standard. It is is supported both in commercial data management systems and by XML document storage products.

Conclusions

The history of database research is strongly tied with that of database management systems: in particular, Codd's work on the relational algebra has been hugely influential in the development of systems. The history of database management systems, in turn, is filled with success stories. Oracle Corporation, historically and primarily concerned with database systems, is a company with 100,000 employees, and one of the 50 largest market capitalizations (Financial Times 2010). Most Web sites critically rely on database software for managing their content, user data, and transaction information, using either major commercial systems such as Oracle, DB2, or SQLServer, or open-source software such as PostgreSQL and the comparatively lightweight MySQL (now owned by Oracle), very popular for simple Web sites and which has captured one third of the market since its initial release in 1995 (Creative System Design 2010).

Through advances in query optimization, indexing structures, but obviously also in hardware, the performance of database management systems has greatly improved over the years: the number of transactions per second in an update-oriented benchmark has thus grown thousandfold over the period 1985–2005 (Gray 2005). On the query side, results of the TPC-H benchmark show that in the past 8 years, the query throughput has been multiplied by 50. It is obviously difficult to determine the parts of performance increase coming from software and hardware advances, but an indication that algorithmic and data structure improvements have had important roles is that the observed price/performance reduction in DBMSs beats Moore's law (Gray 2005).

We have insisted throughout this chapter on two facets of database research: on one hand, models, algorithms, and data structures for the efficiency and effectiveness of existing database systems, and on the other hand, broadly-scoped, often theoretical, sometimes even highly-speculative, research about how data can be modeled, queried, and, more generally, managed. In some cases these lines of work show no evidence of convergence. We have seen earlier in this chapter the example of static-analysis–based query optimization. Decades of work in this area have yielded sophisticated characterizations of query hardness and algorithms for query decomposition and minimization, but they have not been applied in practice. Another example is in spatial databases – although database research in general has had impact in this area, many of the more heavily-researched theories for data models and query algebras (Paredaens et al. 1994) have not influenced practitioners.

The timeframe for acceptance of relational database systems was several years, but perhaps we should be willing to wait many decades to judge subsequent work. The past has shown numerous cases of database research that was considered disconnected from applications for long periods, which has ended up being of use in systems. One of the leading figures in database theory, Moshe Vardi, mentions the example of integrity constraints (Winslett 2006):

> The work on integrity constraints in the late 1970s and early 1980s also received scathing criticism as not being at all relevant to the practice of database systems, only to reemerge later as being of central importance. When you do an exciting piece of research, it is very hard to know whether it will be relevant to the field in the long term. This is true both for theory and for experimental work. The vast majority of theory research results will likely be forgotten, as will be the vast majority of experimental work.

There are certainly missed opportunities in both directions: theoretical results not applied in systems, and practical issues not theoretically modeled and analyzed as they should be.

But of course, DBR has to be measured also by the influence on other areas of computer science. Work on indexing has strong connections with research on data structures and algorithms (compare, for instance, binary search trees and B^{+}-trees). Since the work of Codd, database theory has focused on the complexity of evaluating logics and the expressiveness of logical formalisms: results in this area interact strongly with research in finite model theory, descriptive complexity, and computational complexity (Chap. 15.) Work on the foundations of XML has led to a better understanding of tree automata and tree transducers, a basic subject of study in formal languages. The capacity of DBMSs in handling large quantities of data encouraged the development of techniques to extract patterns and discover knowledge from large databases, a field closely related to DBR known as *data mining*. There are many other examples: The applications of database technology to the management of Web data led to research at the border of databases, information retrieval, and machine learning; database design has links with software engineering, while distributed databases raise numerous questions within networking research.

As for further reading, we have already pointed to a number of references on specific research topics. We now refer to more general works that can be of use to pursue the study of database systems and database research in more depth.

A large number of textbooks cover the design of relational database management systems and review core database research. We mention (Ullman 1988, 1989; Ramakrishnan and Gehrke 2002; Garcia-Molina et al. 2008; Silberschatz et al. 2010), but there are many other excellent examples.

With the notable exception of Ullman (1989), these textbooks deal with database systems rather than with database theory. The main reference in database theory is Abiteboul et al. (1995), which covers foundational aspects of database query languages and data models. An earlier survey, giving a snapshot of theoretically-oriented research at the time, is Kanellakis (1990). Database theory is strongly tied to finite-model theory (Libkin 2004), and the connection between the two is highlighted in Vianu (1996).

Acknowledgments We would like to thank Serge Abiteboul, Georg Gottlob, Evgeny Kharlamov, Yann Ollivier, as well as the editor of this book, Edward K. Blum, for valuable feedback about the content of this chapter.

References

Serge Abiteboul and Catriel Beeri. On the power of languages for the manipulation of complex values. *VLDB J.*, 4:727–794, 1995.

Serge Abiteboul and Paris C. Kanellakis. Object identity as a query language primitive. *J. ACM*, 45:798–842, 1998.

Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, and Pierre Senellart. *Web Data Management*. Cambridge University Press, 2012.

Andrea Acciarri, Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, Mattia Palmieri, and Riccardo Rosati. QuOnto: Querying ontologies. In *Proc. AAAI*, 2005.

Peter M. G. Apers, Carel A. van den Berg, Jan Flokstra, Paul W. P. J. Grefen, Martin L. Kersten, and Annita N. Wilschut. PRISMA/DB: A parallel main memory relational DBMS. *IEEE Trans. Knowl. Data Eng.*, 4(6):541–554, 1992.

Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The description logic handbook: theory, implementation, and applications*. Cambridge University Press, 2003.

Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Inf.*, 1:173–189, 1972.

Catriel Beeri, Ronald Fagin, David Maier, Alberto Mendelzon, Jeffrey Ullman, and Mihalis Yannakakis. Properties of acyclic database schemes. In *Proc. STOC*, 1981.

Michael Benedikt and Christoph Koch. From XQuery to relational logics. *ACM Trans. Database Syst.*, 34(4):1–48, 2009.

Luc Bouganim, Björn Þór Jónsson, and Philippe Bonnet. uFLIP: Understanding Flash IO patterns. In *Proc. CIDR*, 2009.

Wolfgang Breitling. Histograms – myths and facts. In *Proc. Hotsos Symposium on Oracle System Performance*, 2005.

Andrea Calì, Georg Gottlob, and Andreas Pieris. Advanced processing for ontological queries. *PVLDB*, 3(1):554–565, 2010.

Diego Calvanese, Giuseppe De Giacomo, Domenico Lembo, Maurizio Lenzerini, and Riccardo Rosati. Tractable reasoning and efficient query answering in description logics: The L-Lite family. J. Autom. Reasoning, 39(3):385–429, 2007.

R. G. G. Cattell, editor. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann, 1997.

Donald D. Chamberlin and Raymond F. Boyce. SEQUEL: A structured english query language. In *Proc. SIGFIDET/SIGMOD Workshop*, volume 1, 1974.

Donald D. Chamberlin, Morton M. Astrahan, Michael W. Blasgen, James N. Gray, W. Frank King, Bruce G. Lindsay, Raymond Lorie, James W. Mehl, Thomas G. Price, Franco Putzolu, Patricia Griffiths Selinger, Mario Schkolnick, Donald R. Slutz, Irving L. Traiger, Bradford W. Wade, and Robert A. Yost. A history and evaluation of System R. *Commun. ACM*, 24 (10):632–646,1981.

Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *Proc. STOC*, 1977.

Surajit Chaudhuri. An overview of query optimization in relational systems. In *Proc. PODS*, 1998.

Chandra Chekuri and Anand Rajaraman. Conjunctive query containment revisited. *Theor. Comput. Sci.*, 239(2):211–229, 2000.

David L. Childs. Description of a set-theoretic data structure. In *Proc. AFIPS*, 1968.

E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6): 377–387, 1970.

E. F. Codd. Recent investigations in relational data base systems. In *Proc. ACM Pacific*, 1975

Douglas Comer. The ubiquitous B-tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.

Charles D. Cranor, Theodore Johnson, Oliver Spatscheck, and Vladislav Shkapenyuk. Gigascope: A stream database for network applications. In *Proc. SIGMOD*, 2003.

Creative System Design. Databases. http://online.creativesystemdesigns.com/projects/databases.asp, 2010.

Umeshwar Dayal. Of nests and trees: A unified approach to processing queries that contain nested subqueries, aggregates, and quantifiers. In *Proc. VLDB*, 1987.

Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

O. Deux. The story of O2. *IEEE Trans. on Knowl. and Data Eng.*, 2(1):91–108, 1990.

Robert A. Di Paola. The recursive unsolvability of the decision problem for the class of definite formulas. *J. ACM*, 16(2), 1969.

Ronald Fagin. Multivalued dependencies and a new normal form for relational databases. *ACM Trans. Database Syst.*, 2(3):262–278, 1977.

Ronald Fagin. Normal forms and relational database operators. In *Proc. SIGMOD*, 1979.

Ronald Fagin, Jürg Nievergelt, Nicholas Pippenger, and H. Raymond Strong. Extendible hashing – a fast access method for dynamic files. *ACM Trans. Database Syst.*, 4(3):315–344, 1979.

Ronald Fagin, Phokion G. Kolaitis, Renée Miller, and Lucian Popa. Data exchange: semantics and query answering. *Theor. Comput. Sci.*, 336(1):89–124, 2005.

Financial Times. The world's largest companies. Technical Report ft500, Financial Times, 2010.

Raphael A. Finkel and Jon Louis Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Inf.*, 4:1–9, 1974.

Hector Garcia-Molina and Kenneth Salem. Main memory database systems: An overview. *IEEE Trans. Knowl. Data Eng.*, 4(6):509–516, 1992.

Hector Garcia-Molina, Jeffrey D. Ullman, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall Press, second edition, 2008.

Georg Gottlob and Christoph Koch. Monadic Datalog and the Expressive Power of Web Information Extraction Languages. *J. ACM*, 51(1):74–113, 2004.

Georg Gottlob, Christoph Koch, and Reinhard Pichler. Efficient Algorithms for Processing XPath Queries. In *Proc. VLDB*, 2002a.

Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions and tractable queries. *J. Comput. Syst. Sci.*, 64(3):579–627, 2002b.

Naga K. Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. GPUTeraSort: high performance graphics co-processor sorting for large database management. In *Proc. SIGMOD*, 2006.

Goetz Graefe. The Cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18(3): 19–29, 1995.

Jim Gray. A "Measure of transaction processing" 20 years later. *IEEE Data Eng. Bull.*, 28(2): 3–4, 2005.

Stéphane Grumbach and Tova Milo. Towards tractable algebras for bags. In *Proc. PODS*, 1993.

Stéphane Grumbach, Leonid Libkin, Tova Milo, and Limsoon Wong. Query languages for bags: expressive power and complexity. *SIGACT News*, 27(2):30–44, 1996.

Ashish Gupta and Inderpal Singh Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Eng. Bull.*, 18(2):3–18, 1995.

Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proc. SIGMOD*, 1993.

Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. SIGMOD*, 1984.

Marc Gyssens and Dirk Van Gucht. The powerset algebra as a natural tool to handle nested database relations. *J. Comput. Syst. Sci.*, 45(1):76–103, 1992.

Marc Gyssens, Dan Suciu, and Dirk Van Gucht. Equivalence and normal forms for the restricted and bounded fixpoint in the nested algebra. *Information and Computation*, 164 (1):85–117, 2001.

Thomas Haigh. "A veritable bucket of facts": Origins of the data base management system. *SIGMOD Record*, 35(2):33–49, 2006.

Sven Hartmann, Sebastian Link, and Klaus-Dieter Schewe. Functional and multivalued dependencies in nested databases generated by record and list constructor. *Ann. Math. Artif. Intell.*, 46(1–2):114–164, 2006.

Ian Horrocks, Peter F. Patel-Schneider, and Frank van Harmelen. From SHIQ and RDF to OWL: the making of a Web ontology language. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(1):7–26, 2003.

Yannis E. Ioannidis and Viswanath Poosala. Balancing histogram optimality and practicality for query result size estimation. In *Proc. SIGMOD*, 1995.

ISO. *ISO 9075:1987: SQL*. International Standards Organization, 1987.

ISO. *ISO/IEC 9075–4:1996: SQL. Part 4: Persistent Stored Modules (SQL/PSM)*. International Standards Organization, 1996.

ISO. *ISO 9075:1999: SQL*. International Standards Organization, 1999.

Theodore Johnson, S. Muthu Muthukrishnan, Vladislav Shkapenyuk, and Oliver Spatscheck. Query-aware partitioning for monitoring massive network data streams. In *Proc. SIGMOD*, 2008.

Paris C. Kanellakis. Elements of relational database theory. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics (B)*, pages 1073–1156. Elsevier and MIT Press, 1990.

Kothuri Venkata Ravi Kanth, Siva Ravada, and Daniel Abugov. Quadtree and R-tree indexes in Oracle Spatial: a comparison using GIS data. In *Proc. SIGMOD*, 2002.

Arthur M. Keller. Algorithms for translating view updates to database updates for views involving selections, projections, and joins. In *Proc. PODS*, 1985.

Won Kim. On optimizing an SQL-like nested query. *ACM Trans. Database Syst.*, 7(3):443–469, 1982.

Anthony C. Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *J. ACM*, 29(3):699–717, 1982.

C. Koch. On the complexity of nonrecursive XQuery and functional query languages on complex values. *ACM Trans. Database Syst.*, 31(4):1215–1256, 2006.

Isaac Kunen and Dan Suciu. A scalable algorithm for query minimization. Technical Report 02-11-04, University of Washington, 2002.

Neal Leavitt. Whatever happened to object-oriented databases? *Computer*, 33(8):16–19, 2000.

Sang-Won Lee and Bongki Moon. Design of flash-based DBMS: an in-page logging approach. In *Proc. SIGMOD*, 2007.

Maurizio Lenzerini. Data integration: a theoretical perspective. In *Proc. PODS*, 2002.

Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proc. VLDB*, 1996.

Leonid Libkin. Expressive power of SQL. *Theor. Comput. Sci.*, 296(3):379–404, 2003.

Leonid Libkin. *Elements of Finite Model Theory*. Springer, 2004.

Leonid Libkin and Limsoon Wong. Some properties of query languages for bags. In *Proc. DBPL*, 1993.

Leonid Libkin and Limsoon Wong. Query languages for bags and aggregate functions. *J. Comput. Syst. Sci.*, 55(2):241–272, 1997.

Witold Litwin. Linear hashing: A new tool for file and table addressing. In *Proc. VLDB*, 1980.

Diana Lorentz. *Oracle Database SQL Language Reference*. Oracle, 2010.

William C. McGee. The information management system (IMS) program product. *IEEE Annals of the History of Computing*, 31:66–75, 2009.

W. Y. Mok. A comparative study of various nested normal forms. *IEEE Trans. on Knowl. and Data Eng.*, 14(2):369–385, 2002.

Wai Yin Mok, Yiu-Kai Ng, and David W. Embley. A normal form for precisely characterizing redundancy in nested relations. *ACM Trans. Database Syst.*, 21(1):77–106, 1996.

T. William Olle. *The Codasyl Approach to Data Base Management*. John Wiley & Sons, Inc., 1978.

Z. Meral Ozsoyoglu and Li-Yan Yuan. A new normal form for nested relations. *ACM Trans. Database Syst.*, 12(1):111–136, 1987.

M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Database Systems*. Springer, third edition, 2011.

Jan Paredaens and Dick Van Gucht. Converting nested algebra expressions into flat algebra expressions. *ACM Trans. Database Syst.*, 17(1):65–93, 1992.

Jan Paredaens, Jan Van den Bussche, and Dirk Van Gucht. Towards a theory of spatial database queries. In *Proc. PODS*, 1994.

R. L. Patrick. IMS@Conception. *IEEE Annals of the History of Computing*, 31(4):62–65, 2009.

Viswanath Poosala, Yannis E. Ioannidis, Peter J. Haas, and Eugene J. Shekita. Improved histograms for selectivity estimation of range predicates. In *Proc. SIGMOD*, 1996.

J. A. Postley. Mark IV: evolution of the software product, a memoir. *IEEE Annals of the History of Computing*, 20(1):43–50, 1998.

Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. WCB/McGraw-Hill, third edition, 2002.

Arnon Rosenthal and César A. Galindo-Legaria. Query graphs, implementing trees, and freely-reorderable outerjoins. In *Proc. SIGMOD*, 1990.

Manfred Schmidt-Schaubß and Gert Smolka. Attributive concept descriptions with complements. *Artif. Intell.*, 48(1):1–26, 1991.

Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database Systems Concepts*. McGraw-Hill, Inc., sixth edition, 2010.

P. M. Stocker, P. M. D. Gray, and M. P. Atkinson, editors. *Databases – Role & Structure: An Advanced Course*. Cambridge University Press, 1984.

Michael Stonebraker. Technical perspective – one size fits all: an idea whose time has come and gone. *Commun. ACM*, 51(12):76, 2008.

Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. C-store: A column-oriented DBMS. In *Proc. VLDB*, 2005.

Dan Suciu. Bounded fixpoints for complex objects. *Theor. Comput. Sci.*, 176(1–2):283–328, 1997.

V. Tannen, P. Buneman, and L. Wong. Naturally embedded query languages. In *Proc. ICDT*, 1992.

Zahir Tari, John Stokes, and Stefano Spaccapietra. Object normal forms and dependency constraints for object-oriented schemata. *ACM Trans. Database Syst.*, 22(4):513–569, 1997.

Boris A. Trakhtenbrot. Impossibility of an algorithm for the decision problem in finite classes. *American Mathematical Society Translations Series 2*, 23:1–5, 1963.

Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, 1988.

Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 2. Computer Science Press, 1989.

Moshe Y. Vardi. Querying logical databases. In *Proc. PODS*, 1985.

Victor Vianu. Databases and finite-model theory. In *Proc. Descriptive Complexity and Finite Models*, 1996.

W3C. XML path language (XPath). http://www.w3.org/TR/xpath/, November 1999.

W3C. XML path language (XPath) 2.0. http://www.w3.org/TR/xpath20/, January 2007a.

W3C. XQuery 1.0: An XML query language. http://www.w3.org/TR/xquery/, January 2007b.

Marianne Winslett. Moshe Vardi speaks out on the proof, the whole proof, and nothing but the proof. *SIGMOD Record*, 35(1):56–64, 2006.

WinterCorp. 2005 TopTen award winners. Technical report, WinterCorp, 2005.

L. Wong. Normal forms and conservative extension properties for query languages over collection types. *J. Comput. Syst. Sci.*, 52(3):495–505, 1996.

Carlo Zaniolo. A new normal form for the design of relational database schemata. *ACM Trans. Database Syst.*, 7:489–499, 1982.

Chapter 11
Computer Security and Public Key Cryptography

Wayne Raskind and Edward K. Blum

The problem of computer security arises in several contexts in this modern age where most computers are connected to the Internet. Being connected is of course valuable but it also means that the computer is liable to invasion by unfriendly external agents (sometimes called *hackers*) which inject *viruses,* which are insidious programs that can insert themselves into a computer's operating system and cause serious damage in its executable functions and also possibly access confidential files. This raises problems *in computer security*. The problem of virus protection is treated by several software companies who sell their products to the computer owner. In this chapter, we are interested in another aspect of computer security which the computer owner can deal with directly. This is the problem of confidential (or secret) encrypted email. We have already mentioned that Turing worked on cryptography in World War 2.

Secure (Secret) Email

The computer owner usually has access to an email system that operates on a computer network which allows communication with a widespread group of computer users. On most email systems, email messages can be intercepted by unfriendly recipients. Suppose that the owner wishes to communicate with another email user in the system in a manner which excludes all other users, that is, email between these two users should allow secret messages between them. The use of electronic mail (email) to send

W. Raskind (✉)
Arizona State University, Tempe, AZ, USA
e-mail: raskind@asu.edu

E.K. Blum
Department of Mathematics, University of Southern California, Los Angeles, CA, USA
e-mail: blum@usc.edu

E.K. Blum and A.V. Aho (eds.), *Computer Science: The Hardware, Software and Heart of It*, DOI 10.1007/978-1-4614-1168-0_11,

messages is widespread. If two email users, usually called A (for Alice) and B (for Bob), wish to exchange messages which cannot be understood by other email users, they can resort to an *encryption system* which encodes (*encrypts*) messages in some secret format. A widely used type of encryption system is the *public-key cryptosystem* (PKC). A PKC provides two computational algorithms to Alice and Bob:

1. A public *encryption* algorithm E for transforming a message M into an *enciphered* format E(M) which A sends to recipient B (or B sends to A) and which conceals the meaning of M from other email users. For various purposes, E is made available to all users, say by storing its *key* (see below) in a public file
2. A private *deciphering* algorithm D with its own *key* such that Bob (r Alice), the recipient, can easily compute D(EM) to recover M. Stated mathematically,

$$\mathrm{D}(\mathrm{E}(\mathrm{M})) = \mathrm{M} \tag{11.1}$$

Algorithm D is *private* in that its *key* is known only to A and B, so that M can be regarded as a *secure* (or secret) message.

Of course, there are many other scenarios which utilize a PKC. Certainly, we can conceive of a situation involving a group of users who need only to send secret messages to a single supervisory user who has the decryption key.

A much used *public-key cryptosystem* was invented by Diffie and Hellman in 1976. In this type of PKC, the encryption algorithm E is available to all email users; i.e. it is placed in a "public" system file. So any email user can encrypt a message. Furthermore, we shall show that a user who possesses the decyphering algorithm D can verify a sender's *signature* by means of the two keys, as explained below. This allows reliable electronic funds transfers.

In the simple two-person communication scenario, the decryption algorithm D is kept secret from all users other than A and B. E should be a one-to-one onto mapping of the integers (i.e. a permutation) that is easy to compute and satisfies the basic "inverting" equation (11.1) above, where D is the inverse permutation of E. The encryption algorithm E computes a public encryption code of M using the *encryption key* as we shall show. However, the decryption algorithm D has a secret key available only to A and B and D is hard to compute without access to the decryption key. For practical purposes, hackers cannot decrypt the *cyphertext* E(M) in an acceptable time interval. The message M is deemed "secure".

Consider a *cyphertext* C = E(M). Since E is publically known, any *hacker* recipient of the mailed cyphertext C can try a brute force method to retrieve M, that is, the hacker can test all possible text messages TM and compute E(TM) until the hacker finds a TM such that E(TM) = C. Since E is 1:1, this implies that TM = M. Since D is designed to be hard to compute, this brute force method requires so large a number of tests as to be impractical. These properties are possessed by the PKC's invented by Rivest, Shamir and Adleman, as reported in (1978). They are known as *RSA cryptosystems*. They make use of some elementary number theory which can be found in standard textbooks such as Dickson (1929), for example.

RSA Cryptosystems

An RSA system involves three integers, e, d and n. Say that A wishes to send a message (as a string of characters) to B. Using any of several standard techniques, the text message is first encoded as an integer M between 0 and $n - 1$. Thus, M is represented as a string of decimal numerals, digits 0–9 or as a string of 0s and 1s if binary codes are used. For a long message it may be necessary to break the character string of the message text into a sequence of short blocks and encode each block as an integer M. Of course, given the integer M, it is a simple computation to reconstruct the character string of the message. So an RSA system, like most others, operates by encrypting and decrypting numbers M.

Consider the integer $M < n$. In an RSA algorithm, sender A computes the ciphertext $C = E(M)$ by raising M to the (public) power e modulo n, that is, C is the remainder when M^e is divided by n. In number theory notation {Dickson},

$$E(M) \equiv M^e (\text{mod } n) \tag{11.2}$$

Thus, the public encryption *key* is (e, n). To decrypt C, recipient B raises C to the secret power d modulo n. Thus,

$$D(E(M)) \equiv (E(M))^d (\text{mod } n) \tag{11.3}$$

The result in (11.3) is the original message (number code) M. (11.2) and (11.3) are easily computed. How to determine e, d and n so that (11.1) holds? RSA proceeds as follows.

First, compute n as the product of two very large "random" prime numbers p and q. Thus, $n = pq$. Although n is made public, the factors p and q are made known only to A and B. For very large n, it is generally extremely difficult and time-consuming to find its factors p and q. For example, a brute force trial-and-error procedure would simply divide n by each prime number p where $p \leq \sqrt{}\, n$. Suppose $n = 10^{30}$ so that $\sqrt{}\ n = 10^{15}$. Even assuming a gigabyte/s processing speed, it would take 10^6 s (about 10 days) to try all candidates p as factors of n. This is not practical. There is no known algorithm to find the factors of n within a practical time. How does RSA allow A and B to devise an encryption and decryption algorithm defined by (11.2) and (11.3) and satisfying (11.1)?

First, they find a large random integer d which is relatively prime to $(p - 1)(q - 1)$. For example, d can be chosen so that

$$\gcd(d, (p - 1)(q - 1)) = 1, \tag{11.4}$$

where gcd(x, y) is the greatest common divisor of two numbers x and y. Then the integer e is chosen to be the "multiplicative inverse" of d modulo $(p - 1)(q - 1)$, that is, we can choose e to satisfy

$$e \cdot d \equiv 1 \mod ((p - 1)(q - 1)). \tag{11.5}$$

As we shall see. this implies that E and D are inverse permutations of each other, that is,

$$\mathrm{D(E(M)) = M \text{ and } E(D(M)) = M \text{ for all integers } M.} \tag{11.6}$$

To prove that (11.6) holds for any number M when e, d, and n are chosen as above, RSA use some classical basic number theory as set forth in Dickson (1929) for example.

Let $\Phi(n)$ be the classical Euler totient function whose value is the number of positive integers less than n which are relatively prime to n (i.e. have no common divisors with n). For $n = p$ a prime, obviously $\Phi(p) = p - 1$. For $n = pq$, as in our case, we see easily that

$$\Phi(n) = \Phi(p)\Phi(q) = (p - 1)(q - 1) = n - (p + q) + 1.$$

The choice of d as relatively prime to $(p - 1)(q - 1)$, implies that d has a multiplicative inverse e in the *ring* of integers modulo $\Phi(n)$ (This is a bit of elementary algebra). This means we can find e such that

$$e \cdot d \equiv 1 \mod \Phi(n), \text{ that is, } e \cdot d = k\,\Phi(n) + 1 \text{ for some integer k.} \tag{11.7}$$

By the Euler-Fermat theorem Dickson (1929), for any integer M that is relatively prime to n we have

$$M^{\Phi(n)} \equiv 1(\mathrm{mod}\ n). \tag{11.8}$$

For the above choices (11.4) and (11.5) for d and e and the definitions (11.2) and (11.3) for E and D, it follows that

$$D(E(M)) \equiv (E(M))^{d} \equiv (M^{e})^{d}(\mathrm{mod}\ n) = M^{e\cdot d}(\mathrm{mod}\ n)$$

$$E(D(M)) \equiv (D(M))^{e} \equiv \left(M^{d}\right)^{e}(\mathrm{mod}\ n) = M^{e\cdot d}(\mathrm{mod}\ n).$$

By (11.7),

$$M^{e\cdot d} \equiv M^{k\Phi(n)+1}(\mathrm{mod}\ n).$$

By the Euler-Fermat formula (11.8), replacing n by p, it follows that for any M such that p does not divide M,

$$M^{p-1} \equiv 1(\mathrm{mod}\ p). \tag{11.9}$$

But $M^{(p\ -\ 1)(q\ -\ 1)\ +\ 1} = M^{\ \Phi(n)}\ M$. Therefore, for any k, (11.8) implies

$$M^{k\Phi(n)+1} \equiv M(\mathrm{mod}\ p). \tag{11.10}$$

But (11.10) obviously holds also if $M \equiv 0 \pmod p$; i.e. if M divisible by p. So it holds for all M.

A similar proof applies to q. Therefore,

$$M^{k\Phi(n)+1} \equiv M(\text{mod } q). \tag{11.11}$$

The two equations (11.10) and (11.11) together with (11.7) imply

$$M^{e\cdot d} \equiv M^{k\Phi(n)+1} \equiv M(\text{mod } n).$$

But this is precisely the number-theoretic equation for (11.6) based on (11.2) and (11.3).

It shows that E and D are inverse permutations of the integers.

RSA procedures for Encryption and Decryption

A RSA system provides numerical algorithms (11.2) and (11.3) for encryption and decryption. In fact, in their paper (Rivest et al. 1978), RSA give some details of procedures for efficient computation of the powers in (11.2) and (11.3).

Computing the cyphertext $E(M) \equiv M^e$ (modn) requires at most $2\log_2 e$ multiplications **and** $2\log_2$ e divisions to encrypt M. The following program scheme does this

```
Step 1. Obtain the binary representation $e = e_k e_{k-1} \ldots e_1 e_0$, where the $e_i = 0$ or1.
Step 2. Set $C = 1$ (initialize C).
    For $i = k, k-1, \ldots, 0$ do the following steps 3 and 4 on C:
Step 3. Set C = remainder of $C^2/n$; i.e. $C \equiv C^2$ (modn)
Step 4. If $e_i = 1$, set C = remainder of CM/n; i.e. $C \equiv CM$ (modn)
Step 5. Halt with $C \equiv M^e$ (modn).
```

A similar program computes the deciphering $D(E(M)) = M$.

Finding Large Prime Numbers p and q

This is an essential part of a practical RSA system. Among some practical suggestions, RSA recommended in their original paper using 100-digit random primes for p and q, which was appropriate for that time, but larger primes are required now to achieve desired security. So n will have 200 digits. One way to do this is to have the computer generate and test 100-digit odd numbers at random until a prime is found. By the prime number theorem [3] this can take $(\ln 10^{100})/2 = 115$ tests. To test a random number b for primality, RSA suggest a probabilistic algorithm due to Solovay and Strassen [4]. This algorithm chooses a random number a from a uniform distribution on the set $\{1, 2, \ldots, b-1\}$ and tests whether both

$$\mathrm{gcd(a, b) = 1 \text{ and } J(a, b) = a^{(b-1)/2}(mod\ b)}, \tag{11.12}$$

where J is the number theoretic Jacobi function (Dickson 1929). If (11.12) holds for 100 random values of a, then b is prime with a high probability. In fact, if b is actually prime, then (11.12) holds for all a. To see this note that for b odd and a $\leq$ b and gcd(a,b) =1, J(a,b) has its value in $\{-1, 1\}$ and can be computed by the following RSA-suggested program:

```
J(a,b) = if a = 1 then 1 else
              if a is even then J(a/2, b)$(-1)^{(b -1)/8}$
              else J(bmod a, a) $(-1)^{(a-1)(b-1)/4}$.
```

RSA also suggest that to guard against smart factoring algorithms p and q should differ in length by a few digits and both p−1 and q−1 should have large prime factors. For p−1 to have a large prime factor they generate a large random prime u and take p to be the first prime in the sequence iu +1 for i = 2,4, 6. . ..

Signatures

At the outset, we mentioned the important application for including user electronic *signatures* along with email messages. In certain applications, (e.g. in banking by email), user Bob may need to identify himself by a coded *signature*, S say, where S is a number that uniquely identifies Bob. Suppose Bob wishes to send Alice a message M which he "signs" so that she knows M is really from Bob. One scenario for this is for Alice and Bob to have their own encryption and decryption algorithms, say (E_B, D_B) for Bob and (E_A, D_A) for Alice. Then Bob can compute his signature S for message M by decrypting as

$$S = D_B(M).$$

Bob then encrypts S using Alice's algorithm to compute E_A (S) and sends this in secret to Alice. This is possible since E_A is public. Alice then decrypts this message to obtain S by computing D_A (E_A (S)) = S. She knows this is Bob's signature. She then computes E_B (S) = E_B(D_B (M)) = M, again making use of the public nature of E_B. Alice now has the pair (M,S) which is a message-signature pair with the properties of a signed document. Bob cannot later deny signing the document since only he could create S = D_B (M). Furthermore, Alice can prove that E_B (S) = M i.e. that Bob signed the document M. For other security "games" that can play out on the Internet, the reader can consult (Rivest et al. 1978) and the large literature on computer security.

The Diffie-Hellman Protocol

Many public key cryptographic systems use the Diffie-Hellman protocol reported in (Diffie and Hellman 1976) based on a mathematical setting of a finite abelian group G, written additively (Recall from elementary group theory that G has an associative and commutative binary operation +, and an identity element 0 and for every element a an "inverse" element –a such that a + (−a) = 0. For example, take G to be the integers 0, 1,. . .,p−1 with + being addition modulo a prime p). Indeed, suppose the order (number of elements) of G is a large prime number p. It follows easily from elementary group theory that G is a cyclic group of order p, that is, there is an element q in G such that all elements are powers of q. Let q be such a generator; i.e. in additive notation every element in G is a multiple of q. In a communication scenario such as the above with Alice and Bob, Alice picks an integer r between 1 and p. She keeps r secret, computes rq and makes this result a public key. Similarly, Bob picks a secret integer s, computes sq and makes the result his public key. Then both Alice and Bob can compute rsq using the public nature of rq and sq, and their respective secrets r and s since

$$r(sq) = rsq = s(rq).$$

They then use rsq as the private key of their PKC systems. Notice that there is no need to communicate the key rsq. This eliminates one of the undesirable features of older cryptosystems: no courier is needed to set up the system. Furthermore, an eavesdropper may find rq and sq, these being public, but for large p would find it hard to figure out rsq without knowing r or s. The *discrete log* problem is to compute r or s from the knowledge of rq or sq. The *Diffie-Hellman problem* is to compute rsq given rq and sq. The difficulty of such computations depends very much on the nature of the abelian group G. If G is the additive group Z/pZ, they are easy, using the Euclidean algorithm. For if n is the integer rq, we can efficiently divide n by p and determine the remainder, r. If G is a subgroup of order p of the multiplicative group of a finite field F, then the computations can be done in some cases in sub-exponential time using *index calculus*. The basic idea of the index calculus method is to gather many relations among discrete logs of integers whose factors are prime numbers that are much smaller than the cardinality of F, and use linear algebra to solve for r or s above. There are other finite abelian groups in which the Diffie-Hellman problem is interesting and well-suited to cryptography. Note that we are particularly interested in finite abelian groups because they lend themselves very well to computation.

The El Gamal Public Key Cryptosystem System

One of the simplest public key cryptosystems based on the Diffie-Hellman protocol is El Gamal encryption. With notation as above, suppose Bob has a message, m, that is represented as an element of a cyclic group of prime order p with chosen generator q.

Alice chooses an integer r between 0 and p−2 that she keeps secret and broadcasts rq. Bob picks an "ephemeral" integer s in the same range that is used one and only one time to send this message. He then broadcasts the elements c = sq and d = m + rsq. Alice can then decrypt the message by computing d−rc = m + rsq−rsq = m. An eavesdropper would have to compute r, s or rsq in order to recover m from knowledge of c and d.

Elliptic Curve Cryptography (ECC)

In order to use the Diffie-Hellman protocol to make public key cryptosystems, it is important to work with in an abelian group where specific calculations can be done efficiently, but for which it is difficult to solve the discrete log and Diffie-Hellman problems. An *algebraic group* is a group where the operations may be done by computing with polynomials. Well-known examples of finite algebraic groups include Z/nZ and the multiplicative group of a finite field. But there are other types of algebraic groups that do not immediately come to mind and were discovered as a by-product of other research. In the late eighteenth and early nineteenth century, mathematicians such as Legendre and Abel worked with certain "elliptic" integrals that were impossible to compute in closed form. Examples of these arise if you try to compute the arc length of an ellipse, hence the name. They discovered that these integrals exhibit an abelian group-like structure, and this was one of the themes that spurred the development of group theory in the nineteenth century. Around the same time, mathematicians such as Gauss had made great progress in understanding the solution and classification of quadratic equations in 2 and 3 variables. After this, the next simplest equation is one of the form

$$y^2 = x^3 + ax + b,$$

where a and b are fixed elements of a field F that we will take to be a finite field later. This is called a *Weierstrass equation*. One can reduce finding the roots of any cubic polynomial to one of the form on the right hand side of this equation by a linear change of variables to eliminate the quadratic term. The cubic has a double root if and only if $4a^3 + 27b^2 = 0$. In that case, it is not very difficult to study solutions to the equation above, so we assume from now on that the roots of the right hand side are distinct. The set of solutions to such an equation cannot be parameterized in a similar way as can be done for quadratics, and mathematicians struggled with this for several years before realizing that there were great similarities with the difficulties they faced for elliptic integrals. These two research themes then came together in a particularly fortuitous way. A surprising fact is that the set of solutions to the equation above may be made into an abelian group by a chord and tangent method. Briefly, this goes as follows. Since the right hand side is a cubic, a general line will intersect the set of solutions in three points, say A, B and

C (if the line is tangent, then we count the intersection points with multiplicities). Then the abelian group operation + may characterized by the equation

$$A + B = -C.$$

Thus, A + B + C = O, so that three solutions to the equation "add" to the identity element O of the group if and only if they lie on a line. The identity element *O* is the "point at infinity," where we think of both x and y as being infinite. It is easy to see that this group operation may be performed with polynomials in x and y, and so the set of solutions of this equation in F together with *O* forms an algebraic group called an *elliptic curve*. If we denote the equation by E then we will denote this algebraic group by E(F). If F is the field of complex numbers, then E(F) looks like a torus, or equivalently, the Cartesian product of two circle groups.

Upon first sight, elliptic curves may seem abstruse, but they are actually quite easy to calculate with. For reasons that are still not entirely clear from a theoretical point of view, when F is a finite field, E(F) is very well suited to making cryptographic systems that appear to be very secure and yet can be performed efficiently on low power devices such as smartphones or tablets. This is called *elliptic curve cryptography* (ECC). From now on, assume that F is a finite field such as Z/pZ, the group of integers modulo a prime number p. Any finite field F is a finite extension field of some Z/pZ and thus its cardinality Q is a power of p. Let E be an elliptic curve defined by an equation in x and y with coefficients in F as above. Then E(F) is finite, since there are only finitely many possibilities for x and y in the solutions of the Weierstrass equation above. The Riemann hypothesis for elliptic curves over finite fields gives a range for the size N of E(F) in terms of the cardinality Q of the field F. We have

$$|Q + 1 - N| \leq 2\sqrt{Q}$$

This is a very powerful inequality that has many consequences, the main one being that E(F) cannot be too big or too small. Another consequence is that any large prime number P that divides N is approximately the same bit size as Q. A good cryptographic situation is when there is such a large prime P, for then the Diffie-Hellman problem is expected to be difficult.

ECC is believed to be much more secure than RSA or the cryptographic systems based on the Diffie-Hellman problem for the multiplicative group of a finite field. In fact, it is believed that ECC can provide the same security as RSA using cryptographic primes of about one tenth the bit size. This is important for parties exchanging messages on low power devices. That is why in 2003, NSA signed a licensing agreement with the Canadian company, *Certicom*,[1] to use ECC as one of its primary methods of encryption.

[1] see http://www.certicom.com/index.php/news/6-press-rreleases/314-certicom-sells-licensing-rights-to-nsa, http://www.certicom.com/index.php/2005-press-releases/35-2005-press-releases/267-certicoms-ecc-based-solutions-enable-government-contractors-to-add-security-that-meets-nsa-guidelines-

Certain elliptic curves should be avoided when doing ECC. For example, if E has Q or Q + 1 points (trace 1 resp. trace 0), then there are efficient attacks on ECC using liftings to a local field (as in Smart 1999) and the logarithm resp. pairings to reduce to the Diffie-Hellman problem for the multiplicative group of a finite field (as in Menezes et al. 1993). These reductions can reduce the running time of attack algorithms quite significantly in some cases. For the moment, one can avoid these cases and easily find elliptic curves that are "safe." However, it remains unclear how secure ECC really is and this is the subject of much ongoing research. Index calculus methods do not work well for ECC because (it seems) it is much harder to find relations among discrete logs than in the multiplicative group case. In the latter case, the success of index calculus relies on the fact that the group of nonzero rational numbers under multiplication whose denominators are only divisible by a given finite set of prime numbers S is a finitely generated group whose rank is equal to the number of elements of S. Doing this for elliptic curves seems to involve finding large groups of rational points of elliptic curves over number fields, which is both theoretically and computationally difficult. In general, attempts to attack ECC using index calculus type methods have not yet proven effective. Some believe that the height pairing on elliptic curves provides a sort of "golden shield" (see Koblitz 2000) that protects ECC from such attacks. This may well be the case, but it is not at all clear.

References

Dickson, Leonard Eugene. Introduction to the Theory of Numbers. University of Chicago Press, 1929

Koblitz, Neal. Miracles of the Height Function: A Golden Shield Protecting ECC, 4th workshop on Elliptic Curve Cryptography (ECC 2000). [Slides from a talk.] www.cacr.math.uwaterloo.ca/conferences/2000/ecc2000/koblitz.ps

Menezes A, Okamoto T, Vanstone S, 1993. Reducing elliptic curve logarithms to logarithms in a finite field. IEEE Trans. Inform. Theory 39, 1639–1646

Ronald Rivest, Adi Shamir, and Leonard Adleman. Public Key Cryptography, Communications of the ACM, Feb. 1978.

Smart NP, 1999. The Discrete Logarithm Problem on Elliptic Curves of Trace One. J. Cryptology 12(3), 193–196

Whitfield Diffie and Martin Hellman. New Directions in Cryptography. IEEE Transactions on Information Theory, Nov. 1976.

Chapter 12
Complexity Theory

Alfred V. Aho

Introduction

Complexity theory is the area of the theory of computation that deals with the study and classification of the amount of computational resources required to solve problems. The subject is intellectually exciting and central to the field of computer science as well as to understanding how complex systems outside of computer science behave and compute. Complexity theory is an active area of research, still having some of the deepest unsolved problems in mathematics and computer science.

In this chapter we consider the following quintessential questions of complexity theory:

1. How do we measure the performance of an algorithm?
2. Are some problems harder to solve than others?
3. Why are some problems impossible to solve?
4. Is verifying a solution to a problem easier than finding a solution?
5. Is finding an approximate answer easier than finding an exact one?
6. Can randomization speed up computation?

As a simple example of the second question, we can ask whether it is easier given two one-thousand-digit numbers to multiply them together than given one two-thousand-digit number to find its factors? A lot of contemporary computer security technology assumes factoring is a computationally difficult problem (See Chap. 11). Surprisingly, we will discover that definitive answers to many of these fundamental questions are not yet known.

Our discussion will focus on fundamental results and their significance rather than on details of proofs or complete coverage of the field. The reader is encouraged to consult the excellent textbooks and papers cited in the references to find detailed

A.V. Aho (✉)
Columbia University, New York, NY, USA
e-mail: aho@columbia.edu

E.K. Blum and A.V. Aho (eds.), *Computer Science: The Hardware, Software and Heart of It*, DOI 10.1007/978-1-4614-1168-0_12,

proofs and discussions of other exciting areas of complexity theory such as quantum computing (see Chap. 13) that we didn't have the space to cover in this chapter. For the reader's convenience, we shall briefly review some of the material on Turing machines covered in Chaps. 2 and 3.

Languages and Decision Problems

We start by introducing decision problems, yes-no questions that can be formulated as language membership problems. We can always encode instances of problems as finite-length strings of symbols drawn from some finite alphabet. The alphabet consisting of the two symbols 0 and 1 is sufficient, but is often not notationally clear. There is no loss of generality in assuming a problem is encoded as a text string.

A set of finite-length strings with a common characteristic is called a *language*. Languages may have a finite or infinite number of strings but we always assume that there are a countably infinite number of problem instances.

For example, we can formulate the problem of primality testing as a language membership problem by defining the set PRIMES to be the language consisting of the strings of digits that represent prime numbers. To determine whether a number w is prime we ask whether w is a member of the language PRIMES.

As another example, suppose we want to determine, given a directed graph and a pair of nodes, whether there is a path in the graph from the first node to the second. We can encode each instance of this problem as a string w with three components, (G, x, y), where G represents the graph and x and y represent the nodes. We can then define the language PATH to consist of the set of strings (G, x, y) where G contains a path from x to y. To find out whether there is a path in a given graph G from node x to node y, we ask whether or not the string (G, x, y) is in the language PATH.

We are interested in the computational complexity of algorithms for solving language membership problems. We will be primarily interested in the amount of time and the amount of memory required to solve a problem using some model of computation, measured as a function of the length of the input. For example, if w is a string of n digits and we want to investigate the time complexity of primality testing, we can ask how many primitive algorithmic steps are taken as a function of n to determine whether or not w is a string in the language PRIMES. The number of primitive algorithmic steps would be indicative of the number of instructions that would need to be executed, and hence the amount of time, a computer would take to solve the problem.

Models of Computation

Models of computation are at the heart of complexity theory. A modern digital computer is very complex, so detailed mathematical proofs about the behavior of digital computers are unwieldy and difficult to read and write. In complexity theory,

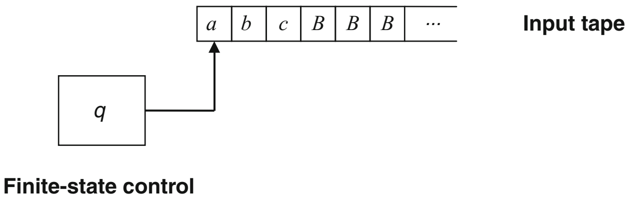


Fig. 12.1 Initial configuration of a single-tape Turing machine

therefore, we use simple mathematical abstractions of computing systems called models of computation in order to make proofs simpler and easier to comprehend. The most important model of computation in computer science is the Turing machine, first proposed by Alan Turing in 1936 (See Chap. 2). We use various kinds of Turing machines and other abstractions as models of computation with which to determine the computational complexity of problems. Although Turing machines are not practical devices, the answers we get using Turing machines serve as indicators of the amount of time or memory programs running on real computers will use to solve problems. The reason for this is that a real computer can be simulated by a Turing machine using roughly the same amount of time or space. We will explain what we mean by "roughly" shortly.

As discussed in Chap. 2, we can think of a deterministic single-tape Turing machine as a finite-state control attached to a tape head that can read and write symbols on the cells of a semi-infinite tape. The tape corresponds to the memory of a computer. Since computer memories are very large, we allow the tape on the Turing machine to be arbitrarily long. Initially, a finite input string of length n is in the leftmost n cells of the tape, an infinite sequence of blanks follows the input string, the tape head is reading the symbol in the leftmost cell, and the finite-state control is in a predefined initial state, as shown in Fig. 12.1.

The Turing machine then makes a sequence of moves. In a move it reads the symbol on the tape under the tape head and consults a transition table in the finite-state control which specifies a symbol to be overprinted on the cell under the tape head, a direction the tape head is to move (one cell to the left or right), and a state to enter next. The tape head never moves past the left end of the input tape. We can think of the transition table as a hardwired program that the Turing machine executes. We will often use the term step as a synonym for move. A move of a Turing machine is analogous to the execution of an instruction of a computer.

Certain states are designated as *accepting* states. If after a finite sequence of moves the Turing machine enters an accepting halting state (one with no next move), it is said to *accept* or *recognize* the input string that was initially on its input tape. If it enters a non-accepting (rejecting) halting state or if it never halts, it does not accept the input string. The language defined by the Turing machine is the set of strings it accepts.

Mathematically, a Turing machine consists of seven components: a finite set of states; a finite input alphabet (not containing the blank); a finite tape alphabet (which includes the input alphabet and the blank); a transition function that maps a state and a tape symbol into a state, tape symbol, and direction (left or right); a start state; an accept state from which there are no further moves; and a reject state from which there are no further moves.

We can characterize the configuration of a Turing machine at a given moment in time by three quantities:

1. The state of the finite-state control,
2. The string of nonblank symbols on the tape, and
3. The location of the input head on the tape.

We shall use the word *configuration* as synonymous with current state, contents of input tape, and location of tape head on tape. A computation of a Turing machine on an input w is a sequence of configurations the machine can go through starting from the initial configuration with w on the tape and terminating (if the computation terminates) in a halting configuration. We say a language is *Turing recognizable* if there is some Turing machine that given any string in the language always halts in the accepting state and given any string not in the language either halts in the nonaccepting state or never halts. The term *recursively enumerable* is often used as a synonym for Turing recognizable.

A language defined by a Turing machine that halts on all inputs is said to be *Turing decidable*. Often the term *recursive* is used as a synonym for Turing decidable. We will use the terms *decider* and *algorithm* as synonyms for a deterministic Turing machine that halts on all inputs.

The fundamental results of computability theory show that there are languages that are not even Turing recognizable and that there are Turing-recognizable languages that are not Turing decidable. Some of these results are proven using the notion of a *universal* Turing machine (Chaps. 2 and 3) – a Turing machine that can simulate the behavior of any given Turing machine on any given input. The universal Turing machine will have on its input tape the specification of the Turing machine to be simulated followed by the given input string. We can think of a universal Turing machine as an ordinary computer that takes a program and data string as input, and then executes the program on that data string.

Computability theory shows that there is a whole host of problems that cannot be solved algorithmically. For example (Chap. 3), we can formally prove that there is no algorithm to determine whether an arbitrary Turing machine will halt on a given input. These results can be used to show that there are an infinite number of problems that cannot be solved by any real computer. A typical undecidability result is that it is impossible to construct a “universal debugger” – a program that will take as input computer programs and determine whether they are free of bugs.

There are many variants of Turing machines including nondeterministic Turing machines and multitape Turing machines. A nondeterministic Turing machine may have a choice of next move from a configuration. Acceptance is defined if and only if there exists a finite sequence of moves that causes the machine to go from an initial configuration to a halting accepting configuration.

The moves of a nondeterministic Turing machine can be represented by a computation tree in which each node is a machine configuration and the children

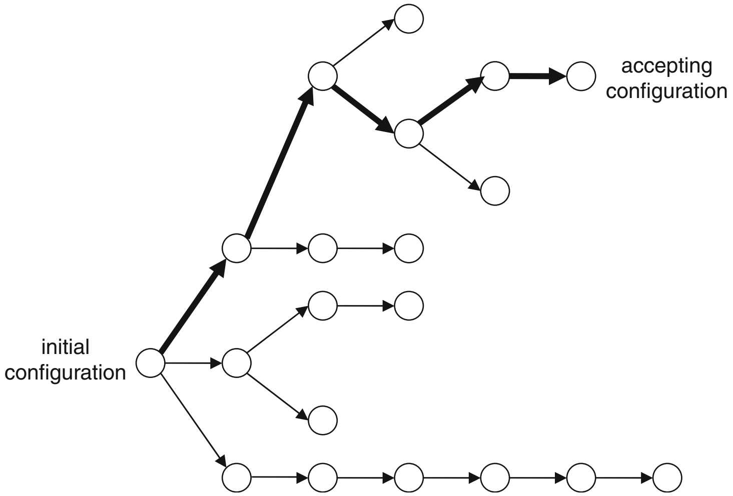


Fig. 12.2 Computation tree showing a path from the initial configuration to an accepting configuration

of a node are the possible next configurations. If there is at least one path in the tree that leads from the initial configuration to a leaf representing an accepting final configuration as shown in Fig. 12.2, then the machine accepts its input. If all paths in the tree from any initial configuration always lead to halting configurations, we call the nondeterministic Turing machine a *decider*.

A nondeterministic Turing machine is a purely mathematical abstraction but as we shall see it is a very powerful tool for classifying the computational difficulty of problems. We should not confuse nondeterminism with randomness. All nondeterminism says is that acceptance of an input occurs if there exists at least one finite sequence of moves that leads from the initial configuration to an accepting configuration. There may be many paths that do not lead to accepting configurations, and there may be infinite paths, but as long as there is one path that terminates in an accepting configuration, the input is accepted. We will discuss models of computation with randomness a little later.

A multitape Turing machine is one that has one input tape and a fixed number of semi-infinite working tapes. The working tapes are initially all blank. Each tape has a tape head and a finite-state control dictates the moves of the machine. Some or all of the tape heads may read or write symbols and move simultaneously during a computational step. A multitape Turing machine may be deterministic or nondeterministic. Acceptance is defined as for a single-tape Turing machine; that is, an input is accepted if there is a sequence of moves on that input that eventually halts in an accepting state.

Nondeterministic and multitape Turing machines do not have any more computational power than single-tape deterministic Turing machines. They can only define the Turing-recognizable languages but they may be able to do computations faster or with fewer tape cells. Unless otherwise qualified, throughout this chapter the term Turing machine will refer to a deterministic single-tape Turing machine.

There are many other models of computation that are equivalent to Turing machines in the sense that they can compute the exact same set of languages. Examples of such models are lambda calculus (Chap. 3 appendix), most programming languages, cellular automata, production systems, and Boolean circuits (Chap. 5). Since much of the theory of computation has been developed using Turing machines, we will use various variants of Turing machines as our primary models of computation throughout this chapter.

Measures of Time Complexity

We focus on the complexity of language membership problems. More specifically, we assume we are given a language L and an input string w. Our problem is to determine whether w is a member of L. We will use two fundamental complexity measures to determine the difficulty of this problem, namely, the time and space needed. The complexity measure depends on the computational model being used to determine whether w is in L.

Time Complexity of Turing Machines

The *time complexity* of a Turing machine is a function $f(n)$ that gives the maximum number of moves the machine could make in processing any input of length n. In complexity theory we are usually interested in the asymptotic running time of an algorithm as it is run on larger and larger inputs. For this reason, we use big-O notation for describing time complexity. If f and g are two functions mapping integers to reals, we say $f(n)$ is $O(g(n))$ if there are positive integers c and m such that $f(n) \leq cg(n)$ for every integer $n \geq m$.

Big-O notation allows us to ignore constant factors and low-order terms so we can focus on the asymptotic growth rate of an algorithm. For example, if $f(n) = 10n^2 + 50n + 1000$, we can say $f(n)$ is $O(n^2)$. With big-O notation we can also ignore the base of logarithms: if $f(n) = \log_2 n$, we can say $f(n)$ is $O(\log n)$.

Given a language L, we say L has time complexity $t(n)$ if L can be recognized by a Turing machine of time complexity $t(n)$. We define the time-complexity class TIME($t(n)$) to be the set of all languages that are decidable by $O(t(n))$-time Turing machines. The term running time is often used as a synonym for time complexity.

Note that our definition of time complexity is a worst-case measure. We have defined the running time of a Turing machine to be maximum number of moves the machine can make on any input of length n. We might also be interested in the expected running time when we can define a probability distribution on all inputs of length n and then take the weighted average of the running times of these inputs to be the expected time complexity. Since determining the expected time complexity of an algorithm can be much more difficult than determining its worst-case time complexity, much of the literature is concerned with worst-case time complexity.

Simulating a Multitape Turing Machine with a Single-tape Turing Machine

We might ask how much faster can we compute using a multitape Turing machine rather than a single-tape Turing machine? The answer is that we can always simulate an $O(f(n))$-time deterministic multitape Turing machine with an $O(f^2(n))$-time deterministic single-tape Turing as follows. The single-tape machine uses its only tape to represent the contents of all of the tapes of the multitape machine by storing the multiple tapes one after each other with an indication of the location of the tape head on each tape and an indication of where each tape ends.

To simulate one move of the multitape machine, the single-tape machine scans its entire tape to determine the symbols under the multiple tape heads. It then makes another pass over its only tape to update the contents of the tapes and the locations of the tape heads. The single-tape machine may need to shift the entire contents of its tape to the right if one of the heads of the multitape machine moves right to scan a blank symbol. Thus each move of the multitape machine can be simulated in $O(f(n))$ steps on the single-tape machine. The entire simulation of the $O(f(n))$-time multitape machine therefore takes $O(f^2(n))$ steps on the single-tape machine.

Time Complexity of Nondeterministic Turing Machines

The model of computation can affect the time or space complexity of a language. The time complexity of a nondeterministic single-tape Turing machine that is a decider is a function $f(n)$ that gives the maximum number of moves the machine can make on any path in the computation tree processing any input of length n. We define the time complexity class NTIME($t(n)$) to be the set of all languages that are decidable by $O(t(n))$-time nondeterministic single-tape Turing machines

There may be an exponential gap between the time complexity of nondeterministic Turing machines and deterministic Turing machines, but no greater. More precisely, every nondeterministic single-tape Turing machine of time complexity $t(n)$ has an equivalent $2^{O(t(n))}$-time deterministic single-tape Turing machine. This result can be shown by a two-stage simulation. We first construct a deterministic multitape Turing machine to systematically trace out all the paths in the computation tree of the nondeterministic machine on an input string of length n in a breadth-first fashion.

Since the nondeterministic Turing machine is a decider, each path in the computation tree from the root to a leaf is of length at most $t(n)$. If c is the maximum number of choices the nondeterministic machine has for any next move, the maximum number of leaves in the computation tree is at most $c^{t(n)}$. Tracing out the entire computation tree can therefore be done in $O(t(n)c^{t(n)}) = 2^{O(t(n))}$ time on the multitape machine.

In the second stage, we can convert the deterministic multitape Turing machine to a single-tape deterministic Turing using the technique we outlined in the previous subsection.

Putting the two stages together, we see that we can first simulate a $t(n)$-time nondeterministic single-tape Turing machine with a $2^{O(t(n))}$-time deterministic multitape Turing machine. We can then simulate this multitape machine with a single-tape machine that takes quadratically more time. But since $2^{O(2t(n))}$ is also $2^{O(t(n))}$, we see that every nondeterministic Turing machine can be simulated with a deterministic Turing machine with at most an exponential increase in time complexity.

The Complexity Classes P and NP

This section presents the complexity classes P and NP that are at the heart of the complexity hierarchy, and introduces the key concept of NP-completeness. One can argue that P, the class of problems that can be solved in polynomial time, became the touchstone for efficient computation in the mid 1960s with the publication of Alan Cobham's paper "The intrinsic computational difficulty of functions" and Jack Edmond's paper "Paths, trees, and flowers" in which they argued that polynomial time was the appropriate measure of efficient computation. It was also around this time that computational complexity became a field of serious study with the early work of Juris Hartmanis, Fred Hennie, and Richard Stearns on complexity-class hierarchies.

The Complexity Class P

The complexity class P is the set of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. P is a good approximation to the class of problems than can be solved on computers in polynomial time because we can simulate virtually every known physical computer that takes $t(n)$ time to solve a problem with a Turing machine that takes $f(n)$ time where $f(n)$ is some polynomial function of $t(n)$. Quantum computers may be an exception.

In complexity theory, we make the assumption that if a problem can be solved in polynomial time, the problem is *tractable*. The reason for this is that algorithms whose growth rates are $O(n)$, $O(n \log n)$, or $O(n^2)$ can easily be run on real computers even on relatively large inputs; algorithms whose growth rates are exponential can only be run on small inputs. It is infeasible to run an algorithm whose time complexity is 2^n on an input whose length is 100.

However, we should note that even though polynomial functions grow less rapidly than exponential functions, it is questionable whether a polynomial-time algorithm such as an n^{100}-time algorithm can be run on a von Neumann computer on inputs whose length is 100. Nevertheless, complexity theory makes a sharp distinction between polynomial and exponential and the class P is at the heart of the time-complexity hierarchy.

Examples of Problems in P

Many common computing problems are in P. Here are three examples, two of which we have already seen.

Recognition of regular sets: Given a regular expression R and a string s, does R match s? This problem can be solved using the McNaughton-Yamada-Thompson (MYT) algorithm in time $O(|R|\cdot|s|)$, where $|R|$ denotes the length (number of symbols) of R and $|s|$ the length of s. To formulate this task as a language membership problem, we can define the language REMATCH to be the set of pairs of strings of the form (R, s) where R matches s. We can use the MYT algorithm to determine whether a pair (R, s) is a member of the language REMATCH in time $O(|R|\cdot|s|)$.

PATH: Given a directed graph G and two nodes x and y in G, is there a path in G from x to y? The problem can be solved in $O(n)$ time using a simple breadth-first search marking algorithm, where n is the number of nodes and edges in G.

PRIMES: Given a string w of digits, is w a prime number? In 2002, Manindra Agrawal, Neeraj Kayal, and Nitin Saxena devised a polynomial-time algorithm for testing whether a number is prime. In 2006, they were honored with the Gödel Prize and the Fulkerson award for this fundamental discovery.

The Complexity Class NP

The complexity class NP is the set of languages that are decidable in polynomial time on a nondeterministic single-tape Turing machine. Interestingly, there is another equivalent way of defining NP using algorithms called verifiers.

A *verifier* for a language L is an algorithm (deterministic Turing machine that halts on all inputs) that takes as input pairs of strings w and c, and accepts w if and only if w is in L and c is a certificate that proves w is in L. The certificate is sometimes called a witness. A polynomial-time verifier is one that runs in polynomial time in the length of w. L is polynomially verifiable if it has a polynomial-time verifier. Note that if L is polynomially verifiable, then for every w there must exist a certificate c whose length is a polynomial function of the length of w.

NP can also be defined as the class of languages that have polynomial-time verifiers. It is easy to show that this definition is equivalent to the original one above.

Given a polynomial-time nondeterministic Turing machine *N* for a language, we can construct a polynomial-time verifier from *N* as follows. The verifier takes as input pairs of strings *w* and *c*. The verifier simulates *N* on input *w* looking at the symbols of *c* as the determiners of the nondeterministic choices *N* should make at each move during its computation. If *c* causes *N* to accept *w*, the verifier accepts *w*. Otherwise, the verifier rejects *w*.

Given a polynomial-time verifier *V* that runs in time n^k for a language, we can construct a polynomial-time nondeterministic. Turing machine for that language from *V* as follows. On an input string *w* of length *n*, the Turing machine nondeterministically creates a certificate string *c* of length at most n^k and simulates the verifier on the pair (*w*, *c*). If *V* accepts, the Turing machine accepts. Otherwise, it rejects.

Examples of Problems in NP

Here are some important examples of languages in NP:

Satisfiability (abbreviated as SAT): SAT is the set of satisfiable Boolean formulas. That is, SAT contains those Boolean formulas for which there is some assignment of truth values to the variables in the Boolean formula that causes the formula to evaluate to TRUE. For example, the Boolean formula

$$w = (x \vee \bar{y} \vee z) \wedge (\bar{x} \vee \bar{y} \vee \bar{z})$$

is satisfiable because the assignment *c* which assigns *x* the value TRUE, *y* the value FALSE, and *z* the value TRUE causes the formula *w* to evaluate to TRUE.

It is easy to construct a polynomial-time verifier for SAT. If a Boolean formula *w* is satisfiable, then it has some satisfying assignment *c*. We can thus present the verifier with the input (*w*, *c*) and all the verifier has to do is evaluate *w* with the assignment *c*. This evaluation can be easily done in time polynomial in the length of *w*. The verifier accepts *w* if and only if *c* is a satisfying assignment.

This example brings out a nice distinction between P and NP. P is the set of languages for which the membership problem can be decided in polynomial time. NP is the set of languages for which membership can be verified in polynomial time.

3SAT is the set of satisfiable conjunctive normal form (CNF) Boolean formulas with exactly three literals per clause. The Boolean formula *w* above is a CNF formula with three literals per clause. 3SAT is in NP for the same reason SAT is in NP – each 3SAT formula has a short, easy-to-verify, satisfying certificate.

HAMPATH: A Hamiltonian path in a directed graph is a path that goes through each node of the graph exactly once. We can define the language HAMPATH that consists of triples of the form (*G*, *x*, *y*) such that *G* is a directed graph with a

Hamiltonian path from x to y. HAMPATH is in NP because we can construct a polynomial-time verifier that takes as inputs pairs of the form $((G, x, y), c)$ where the certificate c represents a Hamiltonian path from x to y. It is easy to verify in polynomial time whether c represents a Hamiltonian path in G.

k-CLIQUE: A k-clique in an undirected graph G is a subgraph H of G with k nodes such that every pair of distinct nodes in H is connected by an edge. The k-CLIQUE language is the set of undirected graphs having a k-clique. A verifier can verify in polynomial time whether a pair (G, k, H) represents an undirected graph G with a k-clique H.

For each of these problems verification is easy. On the other hand, we don't know whether any non-exponential-time algorithms exist for constructing a satisfying assignment, or a Hamiltonian path, or a k-clique. All of the known deterministic algorithms we currently have for constructing satisfying assignments, Hamiltonian paths, and k-cliques all run in exponential time in the worst case.

NP-Completeness

In the early 1970s Stephen Cook and Leonid Levin, independently, made an important discovery that profoundly impacted complexity theory. They showed that certain problems in NP are as hard as any problem in NP. These problems became known as the *NP-complete* problems. The implication of a problem being NP-complete is that if a polynomial-time algorithm were discovered for that problem, then every problem in NP could be solved in polynomial time.

The four problems we mentioned above, SAT, 3SAT, HAMPATH, and k-CLIQUE, are all NP-complete. We just need a few definitions to make this discussion precise.

A function $f(w)$ that maps strings to strings is *polynomial-time computable* if there is a polynomial-time deterministic single-tape Turing machine that started with w on its input tape makes a sequence of moves and always halts with just $f(w)$ on its tape.

A language L is *polynomial-time reducible* to a language M if there is a polynomial-time computable function f such that for every input string w, w is in L if and only if $f(w)$ is in M. The function f is often called a *polynomial-time reduction* of L to M.

A language is *NP-hard* if every language in NP is polynomial-time reducible to it. A language is *NP-complete* if it is NP-hard and in NP.

In the early 1970s Cook and Levin independently showed that SAT is NP-complete. As we indicated above, it is easy to see that SAT is in NP. A verifier can determine in polynomial time if an assignment of truth values satisfies a Boolean formula. The hard part of the proof is to show that every language in NP is polynomial-time reducible to SAT. Suppose we have a nondeterministic Turing machine N that accepts an input string w of length n in time n^k. The essence of the proof is to construct from N and w a Boolean formula whose length is polynomial in n and which corresponds to a computation, and have the formula be satisfiable if and only if *N* accepts *w*. Conceptually the proof is straightforward but the details are involved and will not be further discussed here.

Once we have shown one problem like SAT to be NP-complete, we can prove another problem *p* is NP-complete by showing that *p* is in NP and that SAT is polynomial-time reducible to *p*. 3SAT is also NP-complete and virtually every textbook on complexity theory shows that there is a polynomial-time reduction of SAT to 3SAT.

Thousands of commonly occurring optimization problems have been proven NP-complete. Shortly after the announcement of Cook's result, Richard Karp published an influential paper showing that many important practical optimization problems were NP-complete. Somewhat later, Michael Garey and David Johnson published a comprehensive book cataloguing hundreds of NP-complete problems. This book has become the quintessential collection of NP-complete problems.

Reducibility

In general, a reduction is a way of transforming a problem *A* into another problem *B* such that a solution to *B* can be used to solve *A*. Reducibility is the tool we use to classify the computational complexity of problems. When *A* is reducible to *B*, solving *A* cannot be harder than solving *B*. The notion of reducibility we use throughout this article is called many-one reducibility or occasionally mapping reducibility or Karp reducibility. It is sometimes very difficult to find a polynomial-time reduction from a known NP-complete problem to a suspected NP-complete problem and a number of specialized techniques for performing polynomial-time reductions have been devised to help make such reductions.

Turing reducibility is a more general notion of reducibility that is defined using oracles. An oracle for a language *L* is a mathematical abstraction that will answer whether a string *w* is a member of *L* in a single query. The language *L* does not even have to be Turing recognizable.

An oracle Turing machine is a Turing machine with an attached oracle that can be repeatedly queried by the Turing machine at any step of a computation. We say that a language *A* is *Turing reducible* to language *B* if the language membership problem for *A* can be solved by a Turing machine with an oracle for *B*. The oracle Turing machine may query the oracle for *B* any number of times during a computation. Turing reducibility is a generalization of many-one reducibility, because if *A* is many-one reducible to *B*, then *A* is clearly Turing reducible to *B* – we just call the oracle for *B* once. A polynomial-time Turing reduction is sometimes called a Cook reduction.

The P Versus NP Question

After many decades of trying, the research community has not been able to prove that there is even one language in NP that is not in P. The problem of whether P = NP has become the most celebrated open problem in theoretical computer science. The Clay Mathematics Institute lists it as one of the seven Millenium Prize Problems – problems that the Institute considers as among the most difficult problems in mathematics and for each it offers a one-million dollar award for its solution.

The P versus NP problem even fascinates the mainstream media. For example, within the space of a year John Markoff of the *New York Times* published two articles on the problem, "Prizes Aside, the P-NP Puzzler has Consequences" (October 7, 2009) and "Step 1: Post Elusive Proof. Step 2: Watch Fireworks" (August 16, 2010). The second article reported on a purported proof that P $\neq$ NP that was initially circulated on the Internet. For a few days the purported proof drew a firestorm of attention from the computer science theory community until difficulties were discovered in the proof.

It is possible that P = NP, but many researchers think this is unlikely. If P were, in fact, equal to NP, this would mean that problems like SAT or HAMPATH could be solved by polynomial-time algorithms where today the only known solutions to these problems are exponential techniques. For example, the naive way of determining whether a Boolean formula with n variables is satisfiable is to evaluate the formula on all 2^n truth assignments to its variables, so solving SAT by the obvious brute-force method would take exponential time.

There are some languages in NP that are not known to be NP-complete but for which we know of no polynomial-time algorithm. Perhaps the most famous of these is the graph isomorphism problem. Two undirected graphs G and H are isomorphic if the nodes of H can be renamed so H becomes identical to G. The language ISO consisting of pairs of isomorphic graphs is clearly in NP since it is easy to verify that G and H are isomorphic with an appropriate renaming certificate. However, no one as yet has been able either to prove ISO is NP-complete or to show that there is a polynomial-time algorithm for solving its language membership problem. We don't even know how to prove the complementary language NONISO, consisting of pairs of graphs that are not isomorphic, is in NP.

It is tempting to assume that if P $\neq$ NP, then every problem in NP is either NP-complete or in P. However, such is not the case. Richard Ladner proved that there exist languages in NP that are neither NP-complete nor in P. His result implies that there are infinitely many intermediate classes of languages between P and NP in each of which the languages are polynomial-time reducible to one another. The graph isomorphism problem might be a candidate for a language that is neither NP-complete nor in P.

Space Complexity

Another popular measure of the complexity of a computational problem is the amount of space it requires for a solution. We use the total number of tape cells used by a Turing machine during a computation as the metric of space. This is a rough approximation to the amount of computer memory it would take to solve a problem.

Space Complexity of Turing Machines

If D is a deterministic Turing machine that halts on all inputs, the *space complexity* of D is a function $f(n)$ that gives the maximum number of tape cells the machine uses in processing any input of length n.

If N is a nondeterministic Turing machine whose computation tree is finite on all inputs, the *space complexity* of N is a function $f(n)$ that gives the maximum number of tape cells the machine uses on any path from the root to a leaf of the computation tree in processing any input of length n.

Analogous to the time complexity classes, we can define SPACE($f(n)$) to be the set of languages that can be decided by $O(f(n))$-space deterministic Turing machines, and NSPACE($f(n)$) to be the languages that can be decided by $O(f(n))$-space nondeterministic Turing machines.

With time complexity we observed that it may take exponentially more time for a deterministic decider to recognize a language than a nondeterministic decider. One of the fundamental results in space complexity is Savitch's theorem which states that NSPACE($f(n)$) is contained in SPACE($f^2(n)$), for any function $f(n) \geq n$. This result can be proved using a recursive algorithm that uses $O(f^2(n))$ space to determine whether a nondeterministic Turing machine can go from one configuration to another in $2^{O(f(n))}$ moves.

Sublinear Space Complexity Classes

We can define space complexity classes that are sublinear, e.g., $O(\log n)$, if we use a two-tape Turing machine that has a read-only input tape and another read-write work tape. We measure the space complexity by the number of cells used only on the work tape. For example, the language $\{0^n 1^n \mid n \geq 1\}$ can be decided by an $O(\log n)$-space Turing machine of this nature by counting the number of 0's and 1's separately in binary on the work tape. Only $O(\log n)$ space is needed to store the counters on the work tape.

Two well-studied sublinear space complexity classes are L, the class of languages decidable in logarithmic space on a deterministic two-tape Turing machine, and NL, the class of languages decidable in logarithmic space on a nondeterministic two-tape Turing machine. As with P and NP, we don't know whether NL has more languages than L. Using another type of reducibility, called log-space reducibility, we can define a notion of NL-completeness. We say a language is NL-complete if it is in NL and every other language is NL is log-space reducible to it. We can show the problem PATH is NL-complete.

With NP and NL, we can define the complementary complexity classes coNP and coNL that contain the languages that are the complements of the languages in NP and NL, respectively. We don't know whether NP is different from coNP, but surprisingly we can show that NL and coNL are the same.

The Class PSPACE

There are space analogs for the classes P and NP. We define PSPACE to be the class of languages that are decidable in polynomial space on a deterministic Turing machine and NPSPACE to the class of languages decidable in polynomial space on a nondeterministic Turing machine. Note that by Savitch's theorem, PSPACE $=$ NPSPACE.

The space complexity of a Turing machine limits its time complexity. The time complexity class EXPTIME is the set of all languages decidable by deterministic $O(2^{p(n)})$-time Turing machines, where $p(n)$ is a polynomial function of n. Since a Turing machine computation that halts cannot repeat a configuration, we know that an $f(n)$-space Turing machine must run in $f(n)2^{O(f(n))}$ time; thus PSPACE $=$ NPSPACE $\subseteq$ EXPTIME.

We say a language L is PSPACE-complete if L is in PSPACE and every language in PSPACE is polynomial-time reducible to L.

The PSPACE-complete languages are the most difficult to recognize languages in PSPACE. Quantified Boolean formulas are Boolean formulas containing the quantifiers $\forall$ for "for all" and $\exists$ for "there exists." For example, the quantified Boolean formula $\forall x\varphi$ means that for all values of x the statement φ is true.

A Boolean formula in which each variable appears within the scope of some quantifier is said to be *fully quantified*. We can define the language TQBF to be the set of true fully quantified Boolean formulas. We can use a technique similar to that used to prove Savitch's theorem to show that TQBF is a PSPACE-complete language.

Other problems known to be PSPACE-complete are determining whether a regular expression generates all strings and finding solutions to some games like generalized geography. EXPTIME is interesting in that board games such as generalized chess, checkers, and go are EXPTIME-complete. (A language is EXPTIME-complete if it is in EXPTIME and every language in EXPTIME is polynomial-time reducible to it).

The Class EXPSPACE

The class EXPSPACE is the set of languages that are decidable in exponential space. A language L is EXPSPACE-complete if L is in EXPSPACE and every language in EXPSPACE is polynomial-time reducible to L.

Let us define a class of generalized regular expressions, called regular expressions with exponentiation. If R is a regular expression, define R^k to be R concatenated with itself k times. Let EQREXP be the language consisting of pairs (E, F) of equivalent regular expressions with exponentiation. (Two regular expressions are equivalent if they denote the same set of strings.) We can show that EQREXP is EXPSPACE-complete.

One of the interesting properties of EQREXP is that it is a language that we can prove to be truly intractable; that is, we can prove there is no polynomial-time algorithm to decide it.

NP-Optimization Problems

Let us now turn our attention to optimization problems for which we want to find a best solution in a space of solutions. Here are some examples:

Travelling salesman problem (TSP): Given a list of cities and their pairwise distances, find a cheapest tour (cycle of cities) that goes through each city exactly once.

Minimum vertex cover (MIN-VC): A vertex cover of an undirected graph is a subset of its nodes such that every edge of the graph touches at least one node in the subset. The minimum vertex cover problem is to find a smallest vertex cover for a given undirected graph.

Maximum clique problem (MAX-CLIQUE): Find a largest clique in an undirected graph.

Maximum independent set (MIS): An independent set of an undirected graph is a set of nodes such that no two nodes in the set are connected by an edge. A maximum independent set is a largest independent set.

The first two optimization problems are minimization problems and last two maximization problems.

For each optimization problem, there is a corresponding decision problem. For example, for the travelling salesman problem, we might ask is there a tour that has a cost less than or equal to k?

We will call an optimization problem in which the corresponding decision problem is NP-hard an *NP-optimization problem*. Since the corresponding decision problems for each of the four optimization problems above are NP-complete, these four optimization problems are NP-optimization problems. The question we address in this section is how we might go about trying to find good solutions to NP-optimization problems.

Brute-Force Algorithms

If the problem size is small, we might consider evaluating all possible solutions. This approach only works for small problem sizes. For example, consider the travelling salesman problem with n cities. Suppose we try evaluating all $n!$ tours and picking a cheapest one. Since $n!$ grows exponentially (with $n = 10$, there are 3,628,800 tours; with $n = 20$, there are over 2 quintillion (10^{15}) tours), this approach only works for very small problem sizes.

Heuristic Algorithms

A heuristic is an algorithm that is intended to find quickly a reasonable, but not necessarily optimal solution, to an optimization problem. For example, a heuristic for the travelling salesman problem might start off from a start node and go to a closest node x. It could then go to an unvisited node closest to x, and so on, until it has visited all nodes. This greedy heuristic will produce a Hamiltonian cycle but there is no guarantee that the resulting tour is good let alone optimal. But its advantage is that it runs in linear time.

The travelling salesman problem is one of the most studied optimization problems in computer science and operations research, and dozens of heuristic algorithms for it have been proposed, especially for the Euclidean version of the problem. Practical instances of the problem can be solved by sophisticated heuristics, such as the Lin-Kernighan heuristic, for reasonably large problems but a polynomial-time algorithm that works for all instances of the travelling salesman problem has not been found.

Approximation Algorithms

For some optimization problems we can find algorithms that deliver provably good solutions in a reasonable amount of time. Let us define an approximation algorithm as an efficient algorithm that finds good solutions with a provably good worst-case ratio between the value of the solution found by the algorithm and the true optimum. We say an algorithm is r-approximate for a minimization (maximization) problem if on every input the algorithm finds a solution whose cost is at most r ($1/r$) times the optimum; r is called the performance ratio of the algorithm.

For example, consider the following simple approximation algorithm for the minimum vertex cover problem. Given an undirected graph in which all edges are initially uncovered, pick an uncovered edge and add its endpoint nodes to the vertex cover. Repeat this step until all edges touch the vertex cover. It is easy to show that this algorithm runs in polynomial time and always produces a vertex cover that is no more than twice the size of a smallest vertex cover. This is an example of a *constant factor approximation algorithm* with a performance ratio of two.

Approximation algorithms for optimization problems are an ongoing research area in computational complexity. It turns out that the difficulty of approximating solutions to NP-optimization problems varies greatly. We have just seen that some optimization problems such as vertex cover have constant factor approximation algorithms. At the other extreme, we can show that some optimization problems such as maximal clique have no approximation algorithms that produce solutions within a constant factor of the optimal unless P = NP.

A subject of considerable interest in the design of approximation algorithms is inapproximability. We would like to identify those NP-optimization problems for which the design of an r-approximate algorithm for small r is impossible, unless P = NP. As we shall see, the PCP theorem provides a very powerful tool for proving inapproximability results.

Probabilistic Algorithms

Randomness appears to be inherent in nature. A significant open question in complexity theory is whether randomness can be used to speed up computation.

We can define a probabilistic algorithm using a special type of nondeterministic Turing machine called a probabilistic Turing machine (PTM). A PTM is a nondeterministic Turing machine with two transition functions. At each nondeterministic step of a computation it flips a coin to determine which of the two transition functions to apply.

For a computation on an input w, we assign a probability to each path in the computation tree from the root to a leaf. If the path has k coin-flip steps, the probability of that path is 2^{-k}. The probability that the machine accepts the input w is the sum of the probabilities of each of the accepting paths. The probability that the machine rejects w is one minus the probability that it accepts w.

For $0 \leq \varepsilon < ½$, a probabilistic Turing machine M recognizes language L with error probability ε if

1. w is in L implies Pr[M accepts w] $\geq 1 - \varepsilon$, and
2. w is not in L implies Pr[M rejects w] $\geq 1 - \varepsilon$.

All this says is that the probability that we get the wrong answer by simulating the machine is at most ε.

The time and space complexities of a probabilistic Turing machine are defined in the same way as those for a nondeterministic Turing machine.

Many different polynomial-time randomized complexity classes can be defined using different notions of acceptance. One of the most common is the complexity class BPP, which is the set of languages that are recognizable by probabilistic polynomial-time Turing machines with an error probability of 1/3. BPP stands for bounded-error probabilistic polynomial time.

Amplification

We defined BPP with an error probability of 1/3. In fact, we can change the error probability to any constant strictly between 0 and ½ using a technique called amplification. With amplification we can make the error probability exponentially small. Let $p(n)$ be a polynomial. If M is a polynomial-time PTM that operates with error probability ε, we can construct an equivalent polynomial-time PTM M' that operates with error probability $2^{-p(n)}$ as follows. We have M' run M a polynomial number of times and return the most frequently occurring result. The probability of error decreases exponentially with the number of times M is run. This observation follows from well-known tail bounds on sums of independent random variables from probability theory.

Does Randomness Help?

A major open question is whether BPP = P. In other words, can deterministic Turing machines efficiently simulate all probabilistic Turing machines with at most a polynomial-time slowdown? Or conversely, does randomness increase computational power? Is there a language that can be recognized in polynomial time by a probabilistic Turing machine but not by a polynomial-time deterministic Turing machine.

Research in complexity theory has uncovered unexpected and fascinating connections between the question of whether BPP = P and the hardness of deterministically computing certain functions. If there are very hard functions, then the behavior of such functions appears "random", that is unpredictable, to any deterministic polynomial-time observer. Such functions can be used to generate random bits that are "good enough" for any polynomial-time computation. By "good enough" we mean the bits are indistinguishable from truly random bits.

Randomness is a central feature in interactive proof systems and cryptography.

Interactive Proof Systems

Interactive proof systems have profoundly impacted complexity theory since their definition in 1985 by Shafi Goldwasser, Silvio Micali, and Charles Rackoff. They are widely used in studying cryptography and approximation algorithms and they give us a way of defining a probabilistic analog of the class NP.

We have seen that the languages in NP are those that have short, and easily checkable, proofs of membership. An interactive proof system is a model of computation consisting of two parties, a prover and a verifier, that interact with each other by exchanging messages. The prover is all powerful and can spend an unlimited amount of time constructing proofs of membership. The verifier is a polynomial-time probabilistic Turing machine that checks the proofs given to it by the prover.

We assume the verifier is reliable but the prover can make mistakes. Messages are exchanged between the prover and the verifier until the verifier is convinced the proof is correct. The prover and verifier compute their next message from the message history exchanged to the current point in time.

The objective of the verifier in an interactive proof system is to determine whether an input string w is a member of a given language using the information provided to it by the prover. Instead of having the verifier make probabilistic moves during its computation, we can equivalently give the verifier a random input string r to simulate the effect of the probabilistic decisions. Formally, the verifier's output at each point in time can be modeled by a function $V(w, r, m)$ where

1. w is the input string whose membership in a given language is to be determined,
2. r is the random input given to the verifier, and
3. m is a string consisting of the sequence of messages exchanged between the verifier and prover to the current point in time.

The value of $V(w, r, m)$ is the next message to be sent to the prover, or an indication of whether to accept or reject w.

The prover's behavior can be treated as a function $P(w, m)$ that gives the next message to be sent to the verifier.

The interaction between the prover and the verifier on w and r results in acceptance if there exists a sequence m of alternating messages exchanged between the prover and verifier in which the final message in m is accept.

We assume the length of the verifier's random input and the lengths of the exchanged messages are polynomial functions of n, the length of w. We also assume the number of messages exchanged is also polynomial in n. We define the probability that the interactive proof system accepts w to be the probability of an accepting interaction on a random string r of length polynomial in n.

The Class IP

We can define the complexity class IP as the set of languages L for which there is a polynomial-time verifier function V and an arbitrary prover function P such that for every function P' and string w

1. w is in L implies the probability that an interactive proof system using V and P accepts w with probability $\geq 2/3$, and
2. w is not in L implies the probability that an interactive proof system using V and P' accepts w with probability $\leq 1/3$.

We can use the amplification technique we described earlier to make the error probability of an interactive proof system exponentially small.

It is easy to see that IP contains both the classes NP and BPP. What is surprising, and far less obvious, is that IP = PSPACE. What this implies is that for any language L in PSPACE, a prover can convince a probabilistic polynomial-time verifier that an input string w is in L even though a deterministic Turing machine may spend an exponentially long time proving w is in L.

An example might help to understand better the power of interactive proof systems. Earlier, we mentioned that the language ISO, consisting of pairs of isomorphic graphs, is in NP because for each a pair of isomorphic graphs G and H, there is a short certificate that allows a verifier to determine how the nodes of G can be reordered to make G identical to H.

Now consider the complementary language, NONISO, consisting of pairs of graphs that are not isomorphic. We don't know whether NONISO is in NP since we don't know how to create a short certificate to prove two graphs are not isomorphic. However, an interactive proof system can recognize NONISO executing the following message exchange some number of times.

Consider an input string consisting of two graphs G and H. The verifier randomly selects one of these graphs, randomly reorders its nodes, and sends the reordered graph to the prover. The all-powerful prover then tells the verifier which of G or H was the source of the reordered graph.

If G and H are not isomorphic, the prover would always return the correct answer. However, if G and H are isomorphic, the prover would have to randomly pick G or H and thus answer correctly only half the time. If the prover answers correctly consistently, the verifier becomes more convinced after each iteration that the graphs are not isomorphic. Note that the job of the prover is to convince the verifier after a polynomial number of message exchanges that the two graphs are not isomorphic with high probability.

Probabilistically Checkable Proofs

This section highlights the probabilistically checkable proofs (PCP) theorem, one of the most remarkable results in all of complexity theory. The theorem was discovered in 1992 through the work of a collection of researchers who were investigating why solutions to certain NP-hard optimization problems such as the travelling salesman problem or the independent set problem were hard to approximate. The theorem is remarkable in several ways. It shows that it is possible to transform certain mathematical proofs into a form such that they become checkable by needing to look at only a few probabilistically chosen symbols in the proof. It also shows that computing an approximate solution for some NP-complete optimization problems is as hard as computing the exact solution. Finally, it provides a new characterization for the class NP.

The original proof of the theorem was very complex. Some universities had semester-long courses covering the proof. In 2007 Irit Dinur published a simpler proof, although it too is not that simple. This section explains the theorem but the reader is encouraged to read Dinur's paper for the details of the proof.

Locally Testable Proofs

Suppose we have a certificate for an instance of SAT; that is, we have a proof that shows a Boolean formula is satisfiable. The PCP theorem shows that this proof can be rewritten in such a way that a person can verify the formula by probabilistically selecting a few bits of the proof so that (1) a correct proof will never fail to convince the person that this formula is satisfiable and (2) if the formula is not satisfiable, then the person will reject every purported proof with high probability.

Since SAT is NP-complete, these observations apply to every language in NP. These observations also have implications for checking certain kinds of proofs in mathematics.

PCP Verifiers

A PCP verifier is a generalization of the verifier used in interactive proof systems. A PCP verifier for a language takes as input a string w and a proof π which is just a string of bits. The verifier uses randomness and oracle access to the proof string to decide whether w is a member of the language. Each bit of the proof string can be independently queried by the verifier using a special address tape. If the verifier wants $\pi[i]$, the ith bit of the proof, it writes i on the address tape and then receives $\pi[i]$ as the answer. Note that since the address size is logarithmic in the length of the proof, the PCP verifier can check exponentially long proofs in polynomial time.

We can formalize the definition of the class of languages accepted by PCP verifiers as follows. We use $V^{\pi}(w)$ to denote the output of a PCP verifier V on input w and proof π.

The class PCP[r, q] is defined to contain all languages L for which there is a polynomial-time PCP verifier V that uses $O(r)$ random bits, reads $O(q)$ bits from the proof, and guarantees the following:

1. If w is in L, then there is a proof π such that Pr[$V^{\pi}(w)$ accepts] $= 1$.
2. If w is not in L, then for any proof π, Pr[$V^{\pi}(w)$ accepts] $\leq$ ½.

The PCP theorem states that NP = PCP[log n, 1]. In other words, every NP-optimization problem has a probabilistically checkable proof of logarithmic random complexity and constant query complexity.

Constraint Satisfaction Problems

Satisfiability is an example of a constraint satisfaction problem. In general, a constraint satisfaction problem has three components: a set of variables, a domain of values, and a set of constraints. Constraints are pairs (t, R), where t is an n-tuple of variables and R is a set of n-tuples of values. An evaluation is a mapping v of variables to values, and an evaluation satisfies a constraint $((x_1, x_2, \ldots, x_n,), R)$ if the tuple $(v(x_1)$, $\mathrm{v}(x_2), \ldots, \mathrm{v}(x_n))$ is in R. A solution to a constraint satisfaction problem is an evaluation that satisfies all constraints.

Another way to view the PCP theorem is that it is NP-hard to approximate the maximum fraction of satisfiable constraints of certain constraint satisfaction problems to within some constant factor. For example, the PCP theorem implies that SAT and MIS cannot be approximated efficiently unless P = NP.

For a more specific example, let's consider the problem MAX-3SAT which generalizes SAT. The MAX-3SAT problem is given a Boolean formula in 3CNF to find an assignment of truth values to variables that satisfies the largest number of clauses. Christos Papadimitriou and Mihalis Yannakakis showed that MAX-3SAT is a complete problem for MAXSNP, a class of optimization problems that can be approximated to within a fixed ratio.

It is easy to approximate MAX-3SAT to within a factor of 2 just by considering either the assignment that maps all variables to TRUE or the assignment that maps all variables to FALSE and choosing the assignment that satisfies the most clauses. Since every clause is satisfied by one or the other of the two assignments, one solution will satisfy at least half the clauses.

But how well can we really approximate MAX-3SAT? Johan Håstad showed that for every $\varepsilon > 0$, if there is a polynomial-time $(7/8 + \varepsilon)$-approximation algorithm for MAX-3SAT, then P = NP. Since there are 7/8-approximation algorithms for MAX-3SAT, the implication of this result is that these approximation algorithms are the best that can be obtained (unless P = NP).

In 2001 the Godel Prize was awarded to Sanjeev Arora, Uriel Feige, Shafi Goldwasser, Carsten Lund, Laszlo Lovasz, Rajeev Motwani, Shmuel Safra, Madhu Sudan, and Mario Szegedy for their work on the PCP theorem and its connection to the hardness of approximation.

Relationships Among Complexity Classes

No discussion of complexity theory would be complete without mentioning the known and open containment relationships among the complexity classes. Intuition tells us that if we are given more time or more memory, we should be able to solve larger classes of problems. Complexity theory has time- and space-hierarchy theorems that confirm this intuition subject to certain conditions.

We say a function $t(n)$ is *time constructible* if some $O(t(n))$-time Turing machine exists that always halts with the binary representation of $t(n)$ on its tape when started with an input consisting of n 1's. Most common functions that are at least n log n are time constructible. For example, n log n, n^2, and 2^n are each time constructible. Functions with fractional values such as $n \log_2 n$ are rounded down to the next lower integer.

Likewise, we say a function $s(n)$ is *space constructible* if some $O(s(n))$-space Turing machine exists that always halts with the binary representation of $s(n)$ on its tape when started with an input consisting of n 1's. To show a function is space constructible, we use a Turing machine with a work tape and a separate read-only input tape.

To discuss the time and space hierarchy theorems, we need to define small-o notation for specifying that one function is less than another. If f and g are two functions mapping integers to reals, we say $f(n)$ is $o(g(n))$ if for all $c > 0$ there is a positive number m such that $f(n) < cg(n)$ for every $n \geq m$. What this says is that if $f(n)$ is $o(g(n))$, then the limit of $f(n)/g(n)$ approaches zero as n approaches infinity.

Time-Hierarchy Theorem

The time-hierarchy theorem states that for any time-constructible function $t(n)$ there exists a language L that is decidable in $O(t(n))$ time but not in $o(t(n))/\log t(n)$ time.

The $1/\log t(n)$ factor comes from the fact that we are using a single-tape Turing machine to measure time complexity. The proof is an existence argument, using a diagonalization technique to show that L is not decidable in $o(t(n))/\log t(n)$ time.

Using the time-hierarchy theorem, we can show various containments among time-complexity classes are proper. In particular, we can use the time-hierarchy theorem to show that P is contained in EXPTIME and that there are languages in EXPTIME that are not in P.

Space-Hierarchy Theorem

The space-hierarchy theorem states that for any space-constructible function $s(n)$ there exists a language L that is decidable in $O(s(n))$ space but not in $o(s(n))$ space. We have avoided a $1/\log s(n)$ factor in the theorem by defining a space-constructible function using a Turing machine with a read-only input tape and one additional work tape.

There are many important applications of the space-hierarchy theorem. It allows us to show that there is a fine hierarchy of space-complexity classes within PSPACE: given two real numbers c and d, such that $0 \leq c < d$, we can show

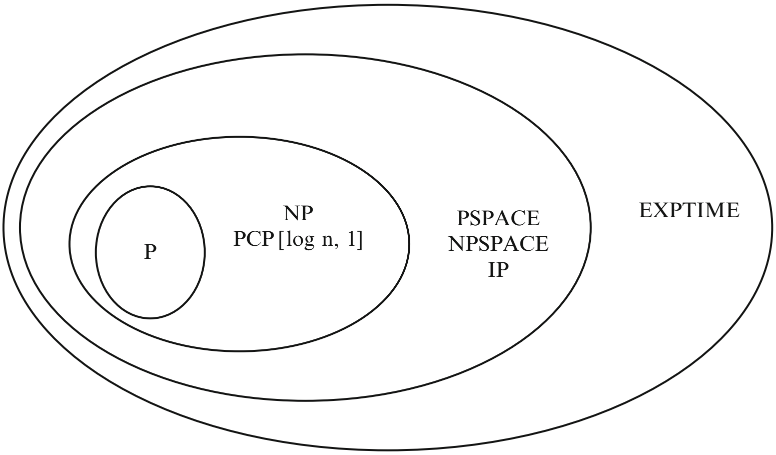


Fig. 12.3 Containment relationships among the fundamental complexity classes

that the complexity class SPACE(n^c) is properly contained within the complexity class SPACE(n^d).

We can also use the space-hierarchy theorem to separate PSPACE from EXPSPACE. We know that PSPACE is contained in EXPSPACE and using the space-hierarchy theorem we can prove that there are languages in EXPSPACE that cannot be decided in PSPACE. In particular, the EXPSPACE-complete language EQREXP that we discussed earlier is thus provably intractable.

The Complexity Zoo

We have already encountered a large number of complexity classes. A lot of effort in complexity theory has been expended in trying to determine the exact containment relationships among these classes.

We know that P $\subseteq$ NP $\subseteq$ PSPACE = NPSPACE = IP $\subseteq$ EXPTIME as depicted in Fig. 12.3. We also know that P $\neq$ EXPTIME because the EXPTIME-complete language EQREXP is provably intractable. (We also know this result from the time-hierarchy theorem.) This implies that at least one of the other containments is proper but at present we don't know which ones.

The fact of the matter is that hundreds of complexity classes have been defined and studied. The website http://qwiki.stanford.edu/index.php/Complexity_Zoo currently lists more than 400 complexity classes, along with some of the most important languages included in them and the known containments among them. Scott Aaronson was the original zoo keeper. But determining which of the many containments among these classes are proper is among the most important open questions in computer science.

Acknowledgement The author is very grateful to Rocco Servedio and Mihalis Yannakakis for many insightful remarks and clarifying comments.

References

Agrawal, M., Kayal, N., and Saxena, N. PRIMES is in P. *Annals of Mathematics 160*, 2 (2004), 781–793.

Aho, A. V., Hopcroft, J. E., and Ullman, J. D. *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

Aho, A. V., Lam, M., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques, and Tools, second edition*, Addison-Wesley, 2007.

Arora, S., and Safra, S. Probabilistic checking of proofs: a new characterization of NP, *Journal of the ACM 45*, 1 (1998), 70–122.

Arora, S., Lund, C., Motwani, R., Sudan, M., and Szegedy, M. Proof verification and the hardness of approximation problems, *Journal of the ACM 45*, 3 (1998), 501–555.

Arora, S., and Boaz, B. *Computational Complexity: A Modern Approach*, Cambridge University Press, 2009.

Babai, L., Fortnow, L., and Lund, C. Nondeterministic exponential time has two-prover interactive protocols, In *Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science* (1990), pp. 16–25.

Babai, L., Fortnow, L., Levin, L., and Szegedy, M. Checking computations in polylogarithmic time, In *Proceedings of the 23rd Annual ACM Symposium on Theory of Computing* (1991), pp. 21–32.

Cobham, A. The intrinsic computational difficulty of functions. In *Proceedings of the International Congress for Logic, Methodology, and Philosophy of Science*, Y. Bar-Hillel, Ed., North-Holland, 1964. pp. 24–30.

Cook, S. A. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Conference on the Theory of Computing* (1971), pp. 151–158.

Dinur, I. The PCP theorem by gap amplification. *Journal of the ACM 54*, 3 (2007).

Edmonds, J. Paths, trees, and flowers. *Canadian Journal of Mathematics 17* (1965), 449–467.

Feige, U., Goldwasser, S., Lovasz, L., Safra, S., and Szegedy, M. Interactive proofs and the hardness of approximating cliques, *Journal of the ACM 43*, 2 (1991), 268–292.

Garey, M. R., and Johnson, D. S. *Computers and Intractability – A Guide to the Theory of NP-completeness*. W. H. Freeman, 1979.

Goldwasser, S., Micali, S., and Rackoff, C. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing* (1989), 186–208.

Hartmanis, J., and Stearns, R. E. On the computational complexity of algorithms. *Transactions of the American Mathematical Society 117* (1965), 285–306.

Håstad, J. Some optimal inapproximability results. *Journal of the ACM 48*, 4 (2001), 105–142.

Hennie, F. C., and Stearns, R. E. Two tape simulation of multitape Turing machines, *Journal of the ACM 13*, 4 (1966), 533–546.

Johnson, D. S. Approximation algorithms for combinatorial problems. *Journal of Computer and Systems Sciences 9* (1974), 256–278.

Karp, R. M. Reducibility among combinatorial problems. In *Complexity of Computer Computations* (1972), R. E. Miller and J. W. Thatcher, Eds., Plenum Press, pp. 85–103.

Ladner, R. On the structure of polynomial time reducibility. *Journal of the ACM 22*, 1 (1975), 155–171.

Levin, L. Universal search problems (in Russian). *Problemy Peredachi Informatsii 9*, 3 (1973), 115–116.

Papadimitriou, C. H. *Computational Complexity*. Addison-Wesley, 1994.

Papadimitriou, C. H., and Yannakakis, M. Optimization, approximation, and complexity classes. *Journal of Computer and Systems Sciences 43*, 3 (1991), 425–440.

Sipser, M. *Introduction to the Theory of Computation, Second Edition*. Thomson, 2006.

Trevisan, L. Inapproximability of combinatorial optimization problems. University of California, Berkeley, 2004.

Turing, A. On computable numbers with an application to the Entscheidungsproblem. In *Proc. London Mathematical Society* 42, pp. 230–265, 1936.

Vazirani, V. *Approximation Algorithms*. Springer, 2003.

Chapter 13
Multivariate Complexity Theory

Michael R. Fellows, Serge Gaspers, and Frances Rosamond

Introduction

Multivariate complexity analysis and algorithm design techniques have developed over many decades, starting from a number of early research themes in Computer Science. The basic insight is that in many situations, one or more secondary measurements of problem instances or computational objectives, beyond the overall input size, govern a problem's computational complexity.

Specific parts of the input or aspects of the problem definition are singled out as the parameter, and the question is whether or not the problem admits an algorithm that is efficient in all but the parameter. Thus, there are positive and negative toolkits – one of techniques for designing efficient parameterized algorithms, and the other to analyze complexity and recognize parameterized intractability. The big advantage is that a single problem can be studied from various points of view, using a variety of possible parameters and their combinations in a multivariate point of view. The intractability shown for a problem with respect to a particular parameter does not mean that parameterized complexity was unsuccessful for the problem, but instead, that more work should be done to reveal more suitable parameters for the problem.

This article attempts to outline some of the key features of the field, with ample references for the interested reader. There are several textbooks and collections of surveys that comprehensively present the field. There were many pioneering investigations of what are now called parameterized algorithms starting in the 1980s. There was concern among the computer science community not only with

M.R. Fellows (✉) • F. Rosamond
School of Engineering and Information Technology, Charles Darwin University,
Casuarina, Australia
e-mail: michael.fellows@CDU.edu.au; frances.rosamond@CDU.edu.au

S. Gaspers
Institute of Information Systems, Vienna University of Technology, Vienna, Austria
e-mail: gaspers@kr.tuwien.ac.at

E.K. Blum and A.V. Aho (eds.), *Computer Science: The Hardware, Software and Heart of It*, DOI 10.1007/978-1-4614-1168-0_13,

```
    TTGACATG CCGGGG---A AACCG
    TTGACATG CCGGTG--GT AAGCC
... TTGACATG -CTAGG---A ACGCG ...
    TTGACATG -CTAGGGAAC ACGCG
    TTGACATC -CTCTG---A ACGCG
```

Fig. 13.1 A partial sample of a multiple alignment among five DNA sequences

whether or not running time was polynomial, but also with whether the exponent could be made constant for every fixed *k*. There were many papers formulating the concept of bounded treewidth, followed by Courcelle's metatheorem showing how to use bounded treewidth for a wide variety of problems, culminating in some sense in efficient algorithms for graphs of bounded treewidth. See Fellows and Langston (1989a), Arnborg et al. (1990), Courcelle (1990), Bodlaender (1993) and others. The corner-stone for parameterized complexity was laid in the foundational monograph by Downey and Fellows in 1998, based on a series of papers in the 1990s, although a completeness program for parameterized intractability was first proposed by Fellows and Langston in 1987, and first attempted in (Abrahamson et al. 1989). Significant impetus for the investigation of parameterized algorithms was lent by developments in the Graph Minors project of Robertson and Seymour (see their Graph Minors series starting about 1983).

These have been followed by books on the subject by Niedermeier (2006) and by Flum and Grohe (2006). A double-special issue of *The Computer Journal* (2008) provides 15 surveys of various aspects of the field. Additional material can be found in proceedings from the two annual international conferences, the "International Symposium of Parameterized and Exact Computation" (IPEC), and the annual workshop WORKER which focuses on kernelization, and in the "Parameterized Complexity Newsletter" edited by F. Rosamond, and archived on the community wiki located at www.fpt.wikidot.com.

A beautiful and useful mathematical theory of multivariate complexity has developed that has become important to applied areas such as bioinformatics, computational social choice, computational reasoning and artificial intelligence – wherever there are NP-hard or otherwise computationally hard problems. We begin our discussion with an example from bioinformatics.

A Concrete Illustration

Most of the problems important to Biologists are *NP*-complete, however problems related to sequence analysis often involve several variables. Some of these may be useful *parameter(s)* which can be used to provide a systematic way of specifying restrictions that may lead to efficient algorithms. We begin the discussion on the importance of parameters with the *multiple sequence alignment* problem (see Fig. 13.1). A multiple sequence alignment displays the similarities of sequences, including gaps where in the process of evolution some parts of the sequence might have been deleted. Sequence alignment has many important uses in Biology – to find relatives of a gene in databases of known genes, to help predict the structure of molecules, to help in the prediction of phylogeny and provide insights into molecular evolution.

There is an efficient polynomial time algorithm to score the quality of a proposed alignment (It might be a function depending on the number of pairwise similarities in the columns of the alignment and the number of gaps inserted to obtain the alignment. However, here we are not concerned about the details of the scoring function).

The decision form of this computational problem is defined as follows.

Multiple Sequence Alignment
Input: Some number of sequences $x_i, i = 1, \cdots, N,$ over an alphabet Σ, and a target score S.
Question: Is there a multiple sequence alignment for the x_i that achieves a score of at least$S$?

This important computational problem is *NP*-hard. It is in *NP*, and so it can be solved in time $O\left(2^{n^c}\right)$, that is, 2 to the exponent a fixed polynomial in n, where n is the number of bits of the input. But this is highly impractical. And, we know that a polynomial-time algorithm is likely not possible. This is bad news.

Let us take a closer look at the problem. For typical inputs that Biologists are concerned with, n, the total number of bits in the input description is often quite *large* – because the sequences can be very long (for example, there are roughly two billion nucleotides in human chromosomes on each of the two strands forming the double-helical DNA molecule). However, there are other relevant measurements which may be *small*, and thus suitable as possible parameters for a parameterized algorithm.

- The *number* N of sequences being aligned (the number of species we are comparing) is often a small number, perhaps less than 10.
- The *size* of the alphabet Σ is small (for DNA sequences, $|\Sigma| = 4$).
- The maximum *distance* Δ between any two of the sequences may be relatively small (i.e., aligning any two of the sequences gives a high score).
- We might also be willing to settle for an alignment that is not optimal, for example, we might be happy with an *approximately optimal alignment*, say, one whose score is within a multiplicative factor of $(1 + \varepsilon)$ of the best possible score. If we could settle for being within 5% of optimal, then $1/\varepsilon$ would be bounded by 20.

In the multivariate approach to complexity analysis and algorithm design, we try to find an algorithm with running time that is polynomial in the input size n, except for an additional charge due entirely to a parameter (which in practice is usually small). If we call our parameter k, we seek an algorithm with a running time of the form $(n^c + f(k))$, where f is some computable function of these secondary numbers and measurements. In our sequence alignment example, the running time we seek would be separated into those two parts and be of the form $O(n^c + f(N, |\Sigma|, \Delta, 1/\varepsilon))$. If the function f were not too bad, for example,

$$f(N, |\Sigma|, \Delta, 1/\varepsilon) = 2^N + 2^{|\Sigma|} + 2^{\Delta} + 2^{1/\varepsilon},$$

then we would have a very useful algorithm, and the Biologists would leave our “Algorithms and Complexity Shop” quite happy.

The purpose of this example is to point to one of the advantages of the parameterized framework – its ability to examine different aspects of the data, and enlist one or more secondary measurements in the design of an efficient algorithm. For an example of a problem that has been well-considered in the parameterized framework, the *NP*-complete Closest Substring problem is important in drug design. On an input of k strings, the problem seeks a length l substring in each of the given strings (a *goal* string) that differs from all the strings by less than some distance d. This problem has been parameterized by d alone, by k alone, by d and k together, and for l, d and k combined. See the survey "Parameterized Complexity and Biopolymer Sequence Comparison" by Cai, Huang, Liu, Rosamond and Song in *The Computer Journal* (2008) for some of the abundant applications of parameterized algorithms in computational biology.

Parameterized Complexity: A Two-Dimensional Theory

This section describes how parameterized complexity may be viewed as a *two-dimensional* complexity theory, and it uses the well-studied problems Vertex Cover and Dominating Set. The Vertex Cover problem is one of the six classic *NP*-complete problems discussed by Garey and Johnson in their famous work on intractability, and because its structure is so simple and elegant, it has played an important role in the development of parameterized algorithms. First we notice that classical complexity theory is based only one measurement, the total number of bits n of the input description (see chapter by Aho in this book). This is essentially a "one-dimensional" framework. Parameterized complexity, which explicitly takes further measurements, can be considered a "two-dimensional" framework.

The definition of a *parameterized decision problem* requires us to specify three things: (1) what is a valid input (2) what is considered the parameter, and (3) what is the question. To illustrate, we define the following two basic parameterized decision problems about graphs. They are illustrated in Fig. 13.2.

Vertex Cover
Input: A graph $G = (V, E)$ and a positive integer k.
Parameter: k.
Question: Does G have a vertex cover of size at most$k$? (A *vertex cover* is a set of vertices $V' \subseteq V$ such that for every edge $uv \in E$, $u \in V'$ or $v \in V'$.)

Dominating Set
Input: A graph $G = (V, E$ and a positive integer k.
Parameter: k.
Question: Does G have a dominating set of size at most k? (A *dominating set* is a set of vertices $V' \subseteq V$ such that every vertex in V/V' has a neighbor in V'.)

Considered classically (that is, just ignoring the parameter specification), both problems are *NP*-complete, and thus unlikely to admit polynomial-time algorithms.

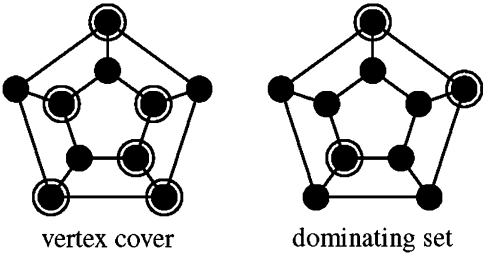


Fig. 13.2 Two vertex set problems with very different complexities. Both problems are *NP*-complete, but *Vertex Cover* is *FPT* parameterized by the size of the solution, while *Dominating Set* is W-hard

Table 13.1 The ratio $\frac{n^{k+1}}{2^k \cdot n}$ for various values of n and k

	$n = 50$	$n = 100$	$n = 150$
$k = 2$	625	2,500	5,625
$k = 3$	15,625	125,000	421,875
$k = 5$	390,625	6,250,000	31,640,625
$k = 10$	1.9×10^{12}	9.8×10^{14}	3.7×10^{16}
$k = 20$	1.8×10^{26}	9.5×10^{31}	2.1×10^{35}

In parameterized complexity theory, the parameter in a parameterized problem can be *anything* that seems relevant. Also, a single classical decision problem can be parameterized in an unlimited number of ways. As a parameter, we have here chosen the value that is optimized in the corresponding optimization problem. We refer to such a parameter as the "standard" parameter.

Both of the parameterized problems Vertex Cover and Dominating Set can be solved in polynomial time for every fixed value of k, simply by the brute force algorithm of trying all k-subsets of vertices and checking if any of them is a vertex cover, or a dominating set, respectively. This brute force algorithm runs in time $O(n^{k+1})$. But they have very different parameterized complexity. There is a simple $O(2^k \cdot n)$ algorithm for Vertex Cover discovered in the early 1980s by Burkhard Monien (See Mehlhorn 1984).

There has been an impressive series of ever-faster parameterized algorithms to solve k-Vertex Cover, leading to the current-best algorithm (Chen et al. 2006) that can decide whether a graph G has a vertex cover of size k in $O(1.2738k + kn)$ time and polynomial space.

The best currently known algorithm for Dominating Set is only slightly better than the simple brute-force algorithm of trying all k-subsets. What these two examples show is that while both problems are *NP*-complete and polynomial-time equivalent, the complexity behavior of their standard parameterization is quite different. The following table shows quantitatively how far apart these behaviors are, in numerical terms that would be quite significant in engineering applications of computing with real datasets (Table 13.1).

Historically, the effort to formalize the difference between the parameterized complexity of Vertex Cover and Dominating Set resulted in the following basic definitions.

Definition: Parameterized Language

A *parameterized language* L is a subset $L \subseteq \Sigma * \times \Sigma *$. If L is a parameterized language and $(x, k) \in L$, then we will refer to k as the *parameter* and write n for the total input size, i.e., $n = |(x, k)|$.

It makes no difference to the theory, and it is occasionally more convenient to consider that k is an integer, or equivalently to define a parameterized language to be a subset of $\Sigma * \times \mathbb{N}$. In particular, the parameter may be non-numerical and there are many natural examples for this. A parameter can also be an aggregate of various kinds of information, as in the example of MULTIPLE SEQUENCE ALIGNMENT.

The central notion of parameterized complexity theory is *fixed-parameter tractability* (FPT).

Definition: FPT

A parameterized language L belongs to the complexity class FPT if there is a function f such that the membership of (x, k) in L can be determined by an algorithm running in time $f(k) + n^c$, where c is a constant and $n = |(x, k)|$ is the total input size. The definition of the parameterized complexity class FPT is unchanged if the additive definition $f(k) + n^c$ is changed to the multiplicative $f(k) \cdot n^c$. Although the definition of the class FPT does not require any formal restriction on the function f, except that it only depends on k and is independent of n, we tacitly assume that for practical FPT algorithms, f is computable and does not grow too fast.

The following definition provides us with a place to put all those problems that "can be solved in polynomial time for every fixed parameter value k" without making our *central distinction* about whether this "fixed k" ends up in the exponent of the running time or not.

Definition: XP

A parameterized language L belongs to the class XP if there are functions f and g such that the membership of (x, k) in L can be determined by an algorithm running in time $f(k)n^{g^{(k)}}$, where $n = |(x, k)\ |$ is the total input size. The fundamental difference between the FPT and XP algorithmic running times laid the corner-stone of parameterized complexity. It has been shown that FPT is a proper subset of XP (Downey and Fellows 1995a).

Figure 13.3 shows the basic intuition about the definition of FPT, and how it generalizes the classical notion of polynomial time computation. The goal is to confine the combinatorial explosion to a function of a small parameter rather than an explosive function of the total input size.

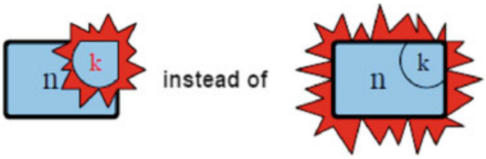


Fig. 13.3 The illustration on the left shows a "controlled explosion", the inevitable combinatorial explosion of an *NP*-hard problem is confined to the parameter "*k*", which may be small. On the right, the explosion encompasses the total input "*n*"

The subject unfolds in two basic complementary projects and associated mathematical toolkits: (1) How to design (and improve) *FPT* algorithms, for parameterized problems that admit them, and (2) How to gather evidence that a parameterized problem probably does not admit an *FPT* algorithm. The next section offers some examples of where parameters may be found in practical problems.

How to Parameterize?

There are many ways that parameters naturally arise in computing. In this section, we give a quite long list of possibilities (some readers may wish to skip ahead to the discussion about complexity workflow at the end of the section), and many more can be found in the references. In parameterized complexity the focus is not on *whether* a problem is intrinsically computationally difficult – the theory starts from the assumption that many interesting and important problems are intractable when considered classically – the focus is on the question: *What makes the problem computationally difficult?* The parameter can be the size of the solution, or some structural aspect of the natural input distribution – and many other things, as illustrated by the following examples.

- *The nesting depth of a logical expression.* ML is a logic-based programming language for which relatively efficient compilers exist. One of the problems the compiler must solve is checking the compatibility of type declarations. This problem is known to be complete for the complexity class *EXP* (deterministic exponential time) (Henglein and Mairson 1991), so the situation appears discouraging from the standpoint of classical complexity theory. However, the implementations work well in practice because the ML Type Checking problem is *FPT* with a running time of $O(2^k \cdot n)$, where n is the size of the program and k is the maximum nesting depth of the type declarations (Lichtenstein and Pneuli 1985). Since normally $k \leq 5$, the algorithm is clearly practical. For many computational problems in diverse areas of applied logic, formula size may be an appropriate parameter.

- *The size of a database query*. Normally the size of a database is huge, but frequently queries are small. If n is the size of a relational database, and k is the size of a query (which of course bounds the number of variables in the query), then determining whether there are objects described in the database that have the relationship described by the query can be solved trivially in time $O(n^k)$. This problem is unlikely to be *FPT* when parameterized this way (Downey et al. 1997; Papadimitriou and Yannakakis 1997). However, it is *FPT* when parameterized by the size of the query and the treewidth of the database (Grohe 2002).
- *The number of voters*. The number of voters in an election may be large, but the number of candidates and the "distance" or "average distance" between votes may be small. Voting systems such as Kemeny, Dodgson, Young, k-approval and others have been shown to be *FPT* with these parameters (Betzler et al. 2009). Parameters are increasingly being used in algorithms for the field of computational social choice. For example, (Shrot et al. 2009) obtained *FPT* algorithms for several parameterizations of classically intractable coalition problems.
- *The number of moves in a game*. The usual computational problem here is to determine if a player has a winning strategy. While most of these kinds of problems are *PSPACE*-complete classically, it is known that some are *FPT* and others are likely not to be *FPT*, when parameterized by the number of moves of a winning strategy. The size n of the input game description usually governs the number of possible moves at any step, so there is a trivial $O(n^k)$ algorithm that just examines the k-step game trees exhaustively. This is potentially a very fruitful area, since games are used mathematically to model many different kinds of situations. These ideas are described in (Scott 2010; Demaine 2001; Fernau et al. 2003).
- *The distance from a guaranteed solution*. Mahajan and Raman (1999) pointed out that for several problems, a solution whose size is a fraction of n may be guaranteed and easy to find. The standard parameterizations of these problems are then trivially *FPT*. A much more reasonable approach is then to parameterize by the size of a solution above or below the guaranteed value. For a simple (and open) example, by the Four Color Theorem and the Pigeon Hole Principle it is always possible to find a four-coloring of a planar graph $G = (V, E)$ where at least one of the colors is used at least $|V|/4$ times. Is it *FPT* to determine if a planar graph admits a four-coloring where one of the colors is used at least $|\mathrm{V}|/4 + k$ times (the parameter is k)?
- *The Hamming weight of a cryptographic key*. Some implementations of public key cryptosystems have considered limiting the size or Hamming weight of keys in order to obtain faster processing times. A cautionary note has been sounded by a result of Fellows and Koblitz (1993) that for every fixed k, with high probability it can be determined in time $f(k)n^3$ whether an n-bit positive integer has a prime divisor less than n^k. If a similar result holds for the Discrete Logarithm problem for exponents of bounded Hamming weight, then the security of some

cryptographic implementations will be compromised. Both problems are trivially solvable in time $O\left(n^{k+c}\right)$, where c is a small constant.

- *The size or structure of variable domains*. Constraint propagation is one of the main tools to solve Constraint Satisfaction Problems. Bessiere et al. (2008) investigate several parameterizations in the propagation of global constraints. Examples are the size of the domains of variables, the number of "holes" in their domains (see also (Gaspers and Szeider 2011)), and the number of symmetry values (see also Walsh 2010). Samer and Szeider (2010) determine the parameterized complexity of the classical Constraint Satisfaction Problem parameterized by the treewidth of several graph representations of CSP instances, combined with several more basic parameters. The SAT problem has been investigated in terms of distance from β-acyclicity by Ordyniak et al. (2010). See (Gottlob and Szeider 2008) for a survey on parameterized complexity in artificial intelligence, constraint satisfaction, and databases.
- *The distance from a given solution*. Another example can be found in local search problems. In the k-Local Search problem for Traveling Salesperson (Marx 2008b), we are given a graph G with positive weights on its edges and a Hamiltonian cycle C in G and the question is whether there is a Hamiltonian cycle which uses at most k edges not used by C with a total edge weight that is smaller than the weight of C. A similar parameterization is used in the Conservative Coloring problem where we are given a graph $G = (V, E)$, a vertex $v \in V$, and a proper k-coloring[1] of $G - v$, and the question is whether G has a proper k-coloring which differs from the original one on at most c places. Both k and c are natural parameters and all parameterizations – by c, by k, and by the combined parameter (c, k) – have been investigated (Hartung and Niedermeier 2010).
- *The number of vertex covers*. Consider the problem of partitioning a graph into parts that are as close in size as possible, that is, their sizes differ by at most one. Such partitions are called "equitable". In problems studied by Suchy (2011), the partitions must satisfy two natural conditions, either every partition is required to induce a connected subgraph, or to induce an independent set. The problems are intractable with respect to the number of partition classes, and so Suchy examines them with respect to various structural measures. The problems remain intractable with respect to the treewidth, the pathwidth and the feedback vertex set number, while becoming tractable with respect to the vertex cover number and the max leaf number.

These are just a few examples to stimulate thinking. The practical world is full of interesting concrete problems governed by parameters of all kinds that are bounded in some small or moderate range. If we can design algorithms with running times

[1] A proper k-coloring of a graph is an assignment of at most k colors to its vertices such that vertices of the same color form an independent set.

like $O(2^k n)$ or $O(2^k + n)$ for these problems, then we may have something really useful. There are now many examples where we can do this for important problems that are *NP*-complete or worse. For the Vertex Cover problem, for example, instances with $k = 200$ have become completely reasonable practical instances, and algorithms for this problem are used in many real-world problems. Michael Langston at the University of Tennessee and Oak Ridge National Laboratory is using Vertex Cover in clustering for problems as varied as the health of the North Seas fisheries, and mouse phenotype analysis; see http://web.eecs.utk.edu/langston/. Michael Langston and Frank Dehne have developed a portal for biologists to use in sequence analysis based on Vertex Cover: http://clustalxp.cgmlab.org/. Associated papers are (Cheetham et al. 2003; Langston et al. 2003; Langston et al. 2004).

Identifying parameters relevant to real-world datasets is something of an art (Niedermeier 2010) and essential to the useful deployment of the multivariate outlook on *NP*-hard problems. In some sense, the search for relevant parameters brings this part of theoretical computer science to the fields of Heuristics and Algorithms Engineering and Artificial Intelligence. In its own terms, this branch of theoretical computer science has developed a distinctive workflow.

The Multivariate Complexity Workflow

Because an *NP*-hard problem can be parameterized in many different ways, the multivariate perspective advances the following two principles that can help in our efforts to discover the source of a problem's hardness, and to design useful algorithms.

- **Principle 1.** When parameterized problems are fixed-parameter tractable, enrich the model by adding more realistic structure.
- **Principle 2.** Hardness proofs should always be "deconstructed", in the search for relevant tractable parameterizations.

An Example of Principle 1: Enriching the Model

Principle 1, asking us to enrich the model for a tractable problem by adding more structure, can be illustrated by the problem of Graph Coloring – for an input graph G, find the minimum k such that there is a proper coloring of G with k colors. This problem is *FPT* when parameterized by the input graph treewidth (Bodlaender 1988). The problem models the important problem of *scheduling*, where the nodes of the graph model the meetings to be scheduled, the colors represent the time blocks, and the edges of the graph represent scheduling conflicts. For example, perhaps For example, perhaps hour-long final exams are being scheduled for courses. If some student is enrolled in both courses, then the exams for these courses should be scheduled in different time blocks.

In reality, there are other sources of scheduling conflicts. There may be some timeblocks where the instructor for a course may be unavailable. A more realistic model of the scheduling problem is Minimum List Coloring, where the input is a graph G, and for each node of the graph, a set of allowable colors that might be assigned to that node (a "list" of acceptable final examination time-blocks that might be assigned for that course). This enriched problem model clearly has greater traction with real world applications. Is this "applications enriched" problem still *FPT* when parameterized by input graph treewidth? The answer is "No," (Fellows et al. 2011) – but the proof that Minimum List Coloring is likely parameterized intractable (by being hard for $W[1]$ – see the next section) is entirely open to review according to Principle 2.

An Example of Principle 2: Deconstructing Hardness

Principle 2 – the deconstruction of hardness proofs, for both *NP*-hardness and $W[1]$-hardness – threatens to make the details of hardness proofs systematically interesting. The key question to ask, that is productive in the multivariate perspective on algorithms and complexity, is:

> Why is the hardness proof *unreasonable*? Why will I never see the images of the transformation that is the basis of the hardness proof in *real-world problem instances*?

The reader who has been exposed to the theory of *NP*-hardness (as might be encountered in an undergraduate education in Information Technology, Computer Science or Operations Research), might wish to persist with this discussion; others might well decide to skip ahead.

The problem Realistic Arbitrage arguably models one of the most fundamental problems in Mathematical Finance. The problem asks whether there is an opportunity to make money from no work, simply by trading currencies. The *realism* in the problem model is that the trades on offer are at specified exchange rates, subject to minimum amounts exchanged at the specified rate. The problem is *NP*-hard. The only known proof of this involves a polynomial-time problem transformation from the Vertex Cover problem where the image instance (for the Realistic Arbitrage problem) of a graph on n vertices, has a number of distinct currencies that is a polynomial function of n. In the real world consideration of arbitrage opportunities, the number of currencies in play is a realistically small parameter, and when the Realistic Arbitrage problem is parameterized by the number of currencies, it is fixed-parameter tractable (Cai and Deng 2003). For more on deconstruction of hardness proofs, see also (Komusiewicz et al. 2009) and Suchy (2011).

A Parameterized Analog of the Cook/Levin Theorem

In the 1970s Stephen Cook and Leonid Levin, independently of each-other, famously discovered that many problems in *NP* were linked together. They discovered that it is possible to transform in polynomial time all problems in *NP* to one particular *NP*-problem, SAT. Furthermore, they discovered that SAT reduces to other problems in *NP*. This section describes a parallel situation for parameterized complexity, where the Clique problem takes the role of SAT. Here, we describe parameterized transformations, and also describe the link to the parameterized analogue of the NDTM Halting Problem, which is trivially *NP*-complete.

It is always possible to parameterize a problem in various ways that are fixed-parameter tractable (for example, any decidable problem is fixed parameter tractable (FPT) parameterized by input length), yet it is not surprising that for many parameterizations of classically hard problems, the resulting parameterized problems apparently are not in *FPT*. This leads to a completeness program based on classes of parameterized problems reasonably presumed to be parameterized intractable, and an appropriate notion of parameterized problem transformation.

Definition: Parametric Transformation

A *parametric transformation* from a parameterized language L to a parameterized language L' is an algorithm that computes from an input consisting of a pair (x, k), a pair (x', k') such that:

1. $(x, k) \in L$ if and only if $(x', k') \in L'$,
2. $k' = g(k)$ is a function only of k, and
3. There is a function f and a constant c such that the computation is accomplished in time $f(k)n^c$, where $n = |(x, k)|$

In first examining the notion of a parametric transformation it can be helpful to see how they differ from ordinary polynomial-time reductions. Consider the Clique problem.

Clique
Input: A graph $G = (V, E)$ and a positive integer k.
Parameter: k.
Question: Does G have a clique of size at least k? (A *clique* is a set of vertices $V' \subseteq V$ such that for every two distinct vertices u and v of V', $uv \in E$.)

Notice that for a graph $G = (V, E)$, a set of vertices $V' \subseteq V$ is a clique in G if and only if $V \setminus V'$ is a vertex cover in the complementary graph G' where two vertices are adjacent if and only if they are not adjacent in G. This gives an easy polynomial-time reduction of the Clique problem to the Vertex Cover problem, transforming an instance $(G = (V, E), k)$ of Clique into an instance (G', k') of Vertex Cover, where $k' = |V| - k$. But this is not a parametric transformation, since k' is not purely a function of k. The evidence is that there is no parametric transformation in this direction between these two problems (although there is a parametric transformation in the reverse direction, either trivially, since Vertex Cover is in *FPT*, or nontrivially by the construction described by Downey and Fellows (1998)).

Downey and Fellows (1995a) have shown a fairly elaborate parametric transformation from the Clique problem to the Dominating Set problem, mapping (G, k) to (G', k') where $k' = 2k$. The evidence is that there is no parametric transformation in the other direction.

The essential property of parametric transformations is that if L transforms to L' and $L' \in FPT$, then $L \in FPT$. On the other hand, if there is evidence that $L \notin FPT$, the same evidence applies to the statement that $L' \notin FPT$.

The parameterized complexity classes $W[t]$, $t = 1, 2, \ldots$, form the W-hierarchy. A parameterized problem L is in $W[t]$ if every instance (x, k) can be transformed by a parametric transformation to a Boolean decision (having one output) circuit, with AND, OR, and NOT gates, of constant depth such that on each path from an input to the output, all but t gates have a constant number of inputs, and $(x, k) \in L$ if and only if the Boolean decision circuit has a satisfying assignment in which at most k inputs are set to 1; see (Downey and Fellows 1995a, 1995b, 1998). We have the following hierarchy of the parameterized complexity classes.

$$FPT \subseteq W[1] \subseteq W[2] \subseteq \ldots \subseteq XP.$$

Under parametric transformations, the following naturally parameterized problems are complete (i.e., they are in the class and hard for the class) for the lowest levels of this hierarchy:

- Vertex Cover is complete for FPT,
- Clique is complete for W[1], and
- Dominating Set is complete for W[2].

Evidence that W-hard problems are probably not fixed-parameter tractable is strengthened by analyzing a parameterization of the Halting Problem. (Chap. 3) Investigations of computability and efficient computability can be classified according to basic forms of the Halting Problem that anchor the discussions. The following form of the Halting Problem is *NP*-complete and essentially sets up the *P versus NP* discussion:

P-Time NDTM Halting Problem
Input: A nondeterministic Turing machine M.
Question: Is it possible for M to reach a halting state in n steps, where n is the length of the description of M?

We generally consider the *P*-Time NDTM Halting Problem to be so unstructured, with nondeterministic computational possibilities, that most computer scientists find the following conjecture compelling. It is widely considered the most important unsolved problem in mathematics and computer science.

Conjecture 1. *There is no polynomial-time algorithm to solve the P-Time NDTM Halting Problem (In other words, $P \neq NP$).*

But, we *can* solve the P-Time NDTM Halting Problem in *exponential time* $O\left(n^{p(n)}\right)$, where p is a polynomial in n, by exploring all possible computational paths of length n, and checking if any of them lead to a halting state. In this sense, the problem is a generic computational embodiment of exponential search. The issue is whether we can get the polynomial $p(n)$ out of the exponent and solve the problem in polynomial time. For this seemingly structureless problem, most people conjecture that this is not possible.

The following fundamental flavor of the Halting Problem establishes the parameterized complexity analog of *P versus NP*, that is, *FPT versus* $W[1]$.

k-Step NDTM Halting Problem
Input: A nondeterministic Turing machine M and a positive integer k.
Parameter: k.
Question: Is it possible for M to reach a halting state in at most k steps when started on an empty input tape?

This problem can be trivially solved in time $O\left(n^k\right)$ by exploring the depth k, n-branching tree of possible computation paths exhaustively.

Conjecture 2. *There is no FPT algorithm to solve the k-Step NDTM Halting Problem.*

Our intuitive evidence for this conjecture is essentially the same as for Conjecture 1. We do not expect to be able to get the parameter k out of the exponent. We do not expect to be able to solve this problem in time $f(k) + n^c$ like VERTEX COVER. In fact, it seems quite difficult to imagine solving the 10-step NDTM Halting Problem in time $O(n^9)$. One could reasonably maintain that our intuitions about Conjecture 1 are exposed in Conjecture 2 with even more compelling directness, although technically Conjecture 2 is stronger (Conjecture 2 implies $P \neq NP$, but the reverse implication is not known to hold).

The k-Step NDTM Halting Problem is complete for the parameterized complexity class $W[1]$ (Downey et al.1994; Cai et al. 1997). Thus, Conjecture 2 implies $FPT[1]$ and $W[1]$ is therefore a strong analog of NP. In particular, Clique and Dominating Set are not in FPT unless Conjecture 2 fails.

We may view the groundbreaking importance of the Cook/Levin Theorem that 3SAT is NP-complete to be in connecting (ultimately) thousands of natural problems (that are NP-hard) with the central generic problem concerning nondeterministic Turing machines, thereby providing powerful intuitive evidence that these problems cannot be solved in deterministic polynomial time.

In the same spirit, intuition suggests that the k-Step NDTM Halting Problem is the most fundamental of the $W[1]$-complete problems, and that it is not fixed parameter tractable. It is natural to regard $W[1]$ as the parameterized analog of NP, and hardness for $W[1]$ as the basic measure for likely parametric intractability. The parameterized complexity analog of the Cook/Levin Theorem is that the

k-STEP NDTM HALTING PROBLEM is fixed parameter tractable if and only if CLIQUE is fixed parameter tractable. The two problems are equivalent with respect to *FPT* reductions. The proof is intricate (Downey and Fellows 1998). Thus, in parameterized complexity theory, the CLIQUE problem plays a role analogous to 3SAT in classical complexity, and is a computationally useful starting point for demonstrations of likely parametric intractability, much as does 3SAT for demonstrations that problems are unlikely to be in P.

Negative Toolkit – INDUCED BICLIQUE Is Hard for $W[1]$

W-hardness results have the "look-and-feel" of NP-hardness results. This section is designed for the reader who may be interested in seeing an example of a parameterized hardness proof. Our example is a reduction from CLIQUE to show that INDUCED BICLIQUE is hard for $W[1]$. We do this in two steps, by first reducing CLIQUE to INDEPENDENT SET, and then INDEPENDENT SET to INDUCED BICLIQUE.

INDEPENDENT SET
Input: A graph $G = (V, E)$ and a positive integer k.
Parameter: k.
Question: Does G have an independent set of size at least k? (An *independent set* is a set of vertices $V' \subseteq V$ such for every two distinct vertices u and v of V', $uv \notin E$.)

INDUCED BICLIQUE
Input: A graph $G = (V, E)$ and a positive integer k.
Parameter: k.
Question: Does G have an induced (k, k)-biclique? (A *biclique* is a set of vertices $V' \subseteq V$ such that V' can be partitioned into two independent sets $A \uplus B$ such that $uv \in E$ for every $u \in A$ and $v \in B$.)

Our method is very similar to proving NP-hardness in that given an instance (G, k) for CLIQUE, we begin by constructing an instance (G', k') for INDEPENDENT SET as follows. The graph G' is the complement of G and $k' := k$. It is easy to see that (G, k) is a yes instance for CLIQUE if and only if (G', k') is a yes instance for INDEPENDENT SET, as any edge of G is a non-edge in G' and vice-versa. Moreover, this is a parametric transformation, as k' depends on k only. Thus, we have the following theorem.

Theorem. *INDEPENDENT SET is hard for* $W[1]$.

As in NP-completeness proofs, we design "gadgets", and must verify "yes iff yes". Now, given an instance $(G = (V, E), k)$ for INDEPENDENT SET, let us construct an instance $(G' = (V', E'), k')$ for INDUCED BICLIQUE as follows. The vertex set V' of G' is a disjoint union of V and I, where I is a set of k new vertices. The edge set E' of G' is obtained from E by adding all pairs uv, where $u \in V$ and $v \in I$. In other words, we have added an independent set of size k to G and added an edge from every vertex of this independent set to every vertex of G. We set $k' := k$.

It remains to show that (1) (G, k) is a yes instance for Independent Set if and only if (2) (G', k') is a yes instance for Induced Biclique.

$(1) \Rightarrow (2)$. Let I' be an independent set of G of size k. Then, $I \cup I'$ is a (k', k')-biclique of G' by the construction of G'.

$(2) \Rightarrow (1)$. Let V' be a (k', k')-biclique of $G'$and let V' be partitioned into two independent sets $A \uplus B$ of size $k' = k$ each. We consider two cases. If $A \subseteq V$, then A is an independent set of size k in G, certifying that (G, k) is a yes instance for Independent Set. Otherwise, $A \cap I \neq \emptyset$. As every vertex of I is adjacent to every vertex of V in G', A contains no vertex from V. Thus, $A = I$. As B is disjoint from A, we obtain that $B \subseteq V$. Thus, B is an independent set of size k in G, certifying that (G, k) is a yes instance for Independent Set. We have the following theorem.

Theorem. *Induced Biclique is hard for* $W[1]$.

The parameterized complexity of the Biclique problem, which is similar to the Induced Biclique problem except that it does not require the sets A and B to be independent, remains open. Despite all the concrete advances that have been accomplished so far, including new techniques, such as reductions from Multicolored Clique (see for example, Fellows et al. 2009a) or breakthrough results such as that the standard parameterization of the Directed Feedback Vertex Set problem is *FPT* (Chen et al. 2008), there is an abundance of still unresolved natural concrete parameterized problems. The next section describes two of the basic techniques for showing tractability.

Positive Toolkit*: FPT* Techniques

The *FPT* technology toolkit generally serves two goals: (1) determine quickly if a problem is *FPT*, and (2) design faster and hopefully practical algorithms. There are a multitude of *FPT* techniques (see "Positive Toolkit" in the next section on Further Reading). In this section we describe in detail two of the most simple and important techniques. These are the methods of *search trees* and *kernelization*. The method of kernelization is so important that there now is an annual workshop devoted to the area. We will illustrate both techniques using the Vertex Cover problem.

Bounded Search Trees

The method of bounded search trees is based on the following strategy. Many combinatorial problems can be solved by recursive algorithms that, for a given instance, compute two or more smaller instances, solve them recursively, and combine the solutions for the smaller instances into a solution for the given instance. The recursive calls of an execution of such an algorithm can be modeled by a tree. Very often, the time that the algorithm spends at each node of the search tree is polynomial. To obtain an *FPT* algorithm, it is then sufficient to bound the size of the search tree by a function of the parameter k.

We demonstrate the bounded search tree method with an algorithm for Vertex Cover and show that it can be solved in time $O(2^k|V(G)|)$. For an instance $(G=(V,E),k)$, the algorithm works as follows. If G has no edge, answer yes. Else, if $k=0$, answer no. Else, choose an edge $uv \in E$.

Any vertex cover V' of G must have $u \in V'$ or $v \in V'$, otherwise the edge uv is not covered. If we select a vertex x to be in the vertex cover, all edges incident to x are covered, and it remains to find a vertex cover of size at most $k-1$ in the graph $G-x$ (the graph obtained from G by removing x and all its incident edges). Thus, the algorithm returns yes if and only if at least one of the recursive calls on $(G-u,k-1)$ and $(G-v,k-1)$ returned yes. As k decreases by one in each recursive call, and the algorithm reaches a leaf of the search tree when $k=0$, the height of the search tree is at most k. As a binary tree of height at most k has at most 2^k leaves, the size of the search tree is $O(2^k)$, and the running time bound follows.

For many search tree algorithms, the branches are less symmetric (in the above example, k decreases by one in both branches of the search tree). Consider the following algorithm for Vertex Cover on an instance $(G=(V,E),k)$. If G has maximum degree at most 2 or $k \leq 0$, solve the problem in polynomial time, and return the answer (a graph of maximum degree at most 2 consists only of paths and cycles, for which an optimal vertex cover can be computed in polynomial time). Otherwise, select a vertex u of degree at least 3. In the first branch, select u to be in the vertex cover and recurse on $(G-u,k-1)$. The second branch considers the choice where u is not in the vertex cover, in which case, all its neighbors need to be in the vertex cover in order to cover the edges incident to u. The algorithm recurses on $(G-N[u],k-d(u))$, where $d(u)$ denotes the degree of u and $N[u]$ denotes the set of vertices containing u and its neighbors. As $d(u) \geq 3$, the number of leaves $T(k)$ of the search tree can be bounded by the recurrence

$$T(k) \leq T(k-1) + T(k-3).$$

Setting $T(0)=1$, this recurrence can be solved by standard mathematical methods and the asymptotic solution is obtained by determining the unique positive root of its characteristic polynomial x^3-x^2-1, which is 1.4655... This shows that Vertex Cover can be solved in time $O(1.466^k|G|)$. More involved analyses consider several cases, leading to a system of recurrences, and measure the size of an instance using a potential function bounded by k. They have led to the currently fastest algorithm with running time $O(1.2738^k+kn)$ (Chen et al. 2006) that we already mentioned earlier.

Kernelization: Reduction to a Problem Kernel

Kernelization is a natural formalization of the notion of polynomial time preprocessing in terms of parameterized complexity. It is known by many names such as "data reduction" or "reduction to a problem kernel". Of course, efficient preprocessing is used in all algorithmic areas, but parameterized complexity gives a natural framework to study how effective the preprocessing is.

The resulting kernelization algorithms can be used prior to almost any approach for solving the problem, such as heuristics or approximation algorithms. The following lemma is trivial.

Lemma. *FPT is equivalent to P-time Kernelization*

A parameterized problem Π is in *FPT* if and only if there is a function g and a polynomial-time (in the input size $|(x,k)|$) transformation that takes (x,k) to (x',k') such that:

1. (x,k) is a yes instance of Π if and only if (x',k') is a yes instance of Π,
2. $k' \leq k$, and
3. $|x'| \leq g(k)$.

We say that we *kernelize* to instances of size at most $g(k)$, and we say that the *kernel* has size $g(k)$. We are interested in finding polynomial-time preprocessing, *or kernelization algorithms* where $g(k)$ is as small as possible.

A kernelization algorithm often consists of a set of (*data*) *reduction rules* that reduce the size of an instance in different situations. We call a reduction rule *sound* if the new instance after an application of the rule is a yes-instance if and only if the original instance is a yes-instance. An instance is *reduced* with respect to a reduction rule if applying the reduction rule to the instance does not change the instance. On real-world problem instances, reduction rules often cascade, and they can be interleaved with bounded search tree branching and pruning techniques, and other methods, in the design of practical algorithms.

Again, using Vertex Cover as an example, we begin with three reduction rules for an instance $(G = (V,E),k)$. They are applied in the order of their appearance.

1. **Isolated Vertex Rule.** If G has a vertex v of degree 0, then remove it and recurse on $(G - v, k)$.

   As v cannot cover any edge, this reduction rule is sound.
2. **High Degree Rule.** If there is a vertex v such that the degree of v (denoted $d(v)$) is greater than k, then add v to the vertex cover and recurse on $(G - v, k - 1)$.

   Any vertex cover that does not contain v, must contain all its neighbors, because the edges incident to v need to be covered. But adding all the neighbors of v to the vertex cover blows the budget, as $d(v){>}k$.
3. **Too Many Vertices Rule.** If Rules 1 and 2 cannot be applied, and if $|V| > k \cdot (k+1)$, then return a trivial no instance (for example, a graph consisting of one edge and parameter 0).

   This rule is sound as $1 \leq d(v) \leq k$ for every vertex v, and every vertex that is not in the vertex cover is the neighbor of at least one vertex of the cover.

We have achieved a kernel with at most $k(k+1)$ vertices. There are other kernelization rules. For example, the "Degree One Rule" (If there is a vertex v with $d(v) = 1$, then put its neighbor u into the solution and recurse on $(G - \{u,v\}, k-1)$) is sound because for any vertex cover V' containing v, the set

$V'' := (V' \setminus \{v\}) \cup \{u\}$ is a vertex cover and $|V''| \leq |V'|$. The Degree One Rule has inspired the more general and powerful *Crown Reduction Rule* (Nemhauser and Trotter 1975; Fellows 2003; Chor et al. 2004), which achieves a kernel on $2k$ vertices.

Usually reduction rules are also used in search tree algorithms in-between branching. This is sometimes called "interleaving" (of the kernelization and the search tree). This usually improves the performance both practically and theoretically. Quite typically an *FPT*-algorithm is formed by a set of rules, some of them being reduction rules (hopefully yielding a kernelization) and some of them being branching rules. Experiments have been conducted to determine how changing the order of reduction rules or of interleaving the rules with the search tree branching can increase efficiency (see the paragraph on Algorithms Engineering in the next section).

Further Reading

Multivariate Complexity Theory is a very active field of research and rapidly growing. As mentioned earlier, the primary references are three books (Downey and Fellows 1998; Niedermeier 2006; Flum and Grohe 2006). The *Parameterized Complexity Newsletter* reports on the latest advances and also contains the Table of Races of the fastest known *FPT* algorithms and the smallest kernels for the most important parameterized problems. The parameterized complexity community wiki at www.FPT.wikidot.com has many pointers to useful resources. In addition to the annual international symposium IPEC with proceedings published by Springer, and WorKer (workshop on kernelization), there is an (almost) annual Dagstuhl seminar (to which one can refer for open problems). There are many journal special issues devoted to various aspects of parameterized complexity (*Discrete Optimization* 2011; *The Computer Journal* 2008 volumes 51 and 53, and *Journal of Computer and System Sciences* 2003, for examples) and numerous dissertations. This section groups some of the key areas of the field, with references for further reading.

The Positive Toolkit. There are many *FPT* techniques in addition to bounded search trees and kernelization. A partial list includes Color-coding, Courcelle's Theorem, Dynamic programming, the Extremal Method, Graph minors, Greedy localization, Iterative compression, Integer linear programming, Modeled crown reductions, Matroid theory, Separators, Tree and branch decompositions, and Well-quasi-ordering (graph minors). Useful are flexible, highly expressive problems, that enable us to solve other problems by reduction to these problems; such as Courcelle's Theorem, the matroid result, 2-SAT Deletion, and constraint satisfaction problems.

The "Ecology of Parameters" explores how one parameter affects the complexity of a different parameterized (or unparameterized problem (Fellows and Rosamond 2007). Many on the above list are demonstrated in an excellent set of slides by Dániel Marx available on his website and at www-sop.inria.fr/mascotte/seminaires/AGAPE. See (Guo et al. 2009) for a summary of results using iterative compression. See an overview of *FPT* techniques in (Guo and Niedermeier 2007) and in (Sloper and Telle 2008).

The Limits of Kernelization. We have seen that the Vertex Cover problem has a kernel on $2k$ vertices. This kernel immediately gives a 2-approximation algorithm for Vertex Cover as well: construct an approximate solution containing all the vertices of the kernel and the vertices forced into the vertex cover by the reduction rules. Running this algorithm for increasing values of k, the first solution for which the kernel is not a trivial no-instance is a 2-approximate solution. This approach works for many problems with linear kernels. Thus, any lower bound on the approximation ratio gives a lower bound on the smallest possible kernel size as well. For Vertex Cover, the inapproximability result of Khot and Regev (2008) implies that it has no kernel with at most ck vertices for any $c < 2$, unless the Unique Games Conjecture fails. This argument rules out linear kernels with certain constant factors.

The kernelization lower bound machinery has been significantly enhanced by frameworks that operate under the complexity assumption that $coNP \not\subseteq NP/poly$. Under this assumption, it has been proved, for instance, that Longest Path has no polynomial kernel (Bodlaender et al. 2009) and that Vertex Cover has no kernel with $O(k^{2-\varepsilon})$ edges, for any $\varepsilon > 0$ (Dell and van Melkebeek 2010). See (Misra et al. 2011) for a survey.

FPT **Optimality.** The $f(k)$ race aims at slower and slower growing functions f in the worst-case running time to solve an *FPT* problem. But how slow can we expect f to grow? Cai and Juedes (2003) show that there is no $2^{o(k)}n^{O(1)}$-time algorithm for Vertex Cover and other parameterized problems unless the Exponential Time Hypothesis (ETH) fails. Cai and Juedes show that Planar Vertex Cover and other problems cannot be solved in time $2^{o\left(\sqrt{k}\right)}n^{O(1)}$ assuming the ETH. See (Flum and Grohe 2006) for an in-depth treatment of this subject.

XP **Optimality**. Similarly, one might wonder, for problems that can be solved in time $n^{g(k)}$, whether one can do much better. What is a limit for the best possible *XP* algorithm? In this line of research, it has been shown that

- There is no $|V|^{o(k)}$-time algorithm for Independent Set unless ETH fails (which would imply that $FPT = M[1]$) (Chen et al. 2005), and that
- There is no $|V|^{o(k)}$-time algorithm for Dominating Set unless $FPT = M[2]$ (Chen et al. 2005).

We refer to (Chen and Meng 2008) for a survey on *XP* Optimality.

Parameterized Complexity and Approximation. As in our introductory example on the Multiple Sequence Alignment problem, one may want to parameterize by $1/\varepsilon$ when the goal is to find a $(1 + \varepsilon)$-approximation for a problem. If an algorithm shows that our problem, parameterized by $1/\varepsilon$, is in *XP*, one speaks of a *PTAS*, a polynomial time approximation scheme. As the degree of the polynomial depends on $1/\varepsilon$, such an algorithm becomes rapidly impractical for reasonably small approximation factors. An *EPTAS* (Efficient *PTAS*) is an *FPT* algorithm for the $1/\varepsilon$-parameterization of our problem, and may exhibit nicer computational properties, even for small approximation ratios. However, we may not expect an *EPTAS* for any problem whose standard parameterization is $W[1]$-hard, since running the *EPTAS* with $\varepsilon = 1/(k+1)$ would solve the problem exactly in *FPT* time.

An *FPT* approximation algorithm is an algorithm with *FPT* running time and an approximation ratio which may depend on the parameter. For example, the Cliquewidth problem, parameterized by the cliquewidth k of the graph, has an *FPT* approximation algorithm with approximation ratio $\left(2^{3k+2}-1\right)/k$ (Oum 2005), whereas the standard parameterization of Independent Dominating Set has no *FPT* approximation algorithm with performance ratio $\varrho(k)$ for any computable function ϱ, unless $FPT = W[2]$ (Downey et al. 2006). See Marx (2008a) for a survey.

Algorithms Engineering. Experiments, implementations, and studies of performance in practice of parameterized algorithms, all help shed more light on aspects of algorithm design such as useful problem structure or the trade-offs between cost/benefits of which reduction rules to apply and in which order. It is not likely that implementation in other fields will take an algorithm "whole-cloth", but instead will pick and choose the reduction rules or branching strategies employed in *FPT* algorithms (Fellows 2002). There has been Dagstuhl Seminar 05301 (2005) on Parameterized Algorithms Engineering. Falk Hüffner has been a leader in parameterized algorithms engineering, especially applied to bioinformatics. See for example, (Hüffner 2009) and (Helwig et al. 2010). Pablo Moscato has been analyzing cancer datasets using parameterized with memetic algorithmic techniques (Rizzi et al. 2010). See also (Tazari and Müller-Hannemann 2009) for sophisticated algorithms engineering of *FPT* algorithms, and see (Fellows et al. 2009b) and (Fomin et al. 2010) for applications to local search.

Parameterized Complexity in Theory Formation. The fine-grained analysis available in the parameterized framework allows analysis that is relevant to fields of science concerned with various natural forms of computation. For example, Iris van Rooij and Todd Wareham have surveyed uses of parameterized complexity analysis in modeling issues relevant to theory-formation in Cognitive Science (van Rooij and Wareham 2008). In the same *Computer Journal* special issue, the survey by Demaine and Hajiaghayi explores some of the multivariate theme (Demaine and Hajiaghayi 2008), and further discussion on the multivariate framework can be found in Fellows (2009), Niedermeier (2010) and Suchy (2011).

Conclusions

Multivariate complexity is in some sense a very old subject in computer science. Practitioners, and even theorists, have paid attention to natural problem parameters, and designed efficient algorithms that take them into account, "since the beginning". It can happen that one offers a practical computing scientist an *FPT* algorithm for an *NP*-hard problem having a natural small parameter and be told: "That's what I already do!" Thus, the field provides a firm theoretical foundation to support existing heuristics and practical computing.

Parameterized complexity has been the opening chapter of a broader exploration, that of multivariate complexity analysis. Tools and methodology in the positive and negative toolkits, kernelization rules that serve particular as well as classes of problems, and lower/upper bound techniques are increasingly being imported and used by other fields. The powerful technology that has been developed allows for a fine-grained exploration of problem structure in the search for effective complexity assessment and algorithm design. Perhaps even more important is the fresh view that the deconstruction of proofs of either *NP*-hardness or *W*-hardness offers a starting point for obtaining new insights into the combinatorial structure of problems.

References

K. Abrahamson, J. Ellis, M. Fellows and M. Mata (1989). On the complexity of fixed-parameter problems. *FOCS 1989*, 210–215.

S. Arnborg, A. Proskurowski, D. Seese (1990). Monadic Second Order Logic, Tree Automata and Forbidden Minors. *CSL 1990*,1–16.

C. Bessiere, E. Hebrard, B. Hnich, Z. Kiziltan, C.-G. Quimper, and T. Walsh (2008). The parameterized complexity of global constraints. *AAAI 2008*, 235–240.

N. Betzler, M. R. Fellows, J. Guo, R. Niedermeier, and F. A. Rosamond (2009). Fixed-parameter algorithms for Kemeny rankings. *Theoretical Computer Science* 410(45), 4554–4570.

H. L. Bodlaender (1988). Dynamic programming on graphs with bounded treewidth. *ICALP 1988*, 105–118.

H. L. Bodlaender: A linear time algorithm for finding tree-decompositions of small treewidth. *STOC* 1993: 226–234.

H. L. Bodlaender, R. G. Downey, M. R. Fellows, and D. Hermelin (2009). On problems without polynomial kernels. *Journal of Computer and System Sciences* 75(8), 423–434.

L. Cai, J. Chen, R. G. Downey, and M. R. Fellows (1997). The parameterized complexity of short computation and factorization. *Archive for Mathematical Logic* 36(4/5), 321–337.

L. Cai, X. Huang, C. Liu, F. Rosamond, and Y. Song (2008). Parameterized complexity and biopolymer sequence comparison. *The Computer Journal* 51(3), 270–291.

L. Cai and D. Juedes (2003). On the existence of subexponential parameterized algorithms. *Journal of Computer and System Sciences* 67(4), 789–807.

M-C. Cai, X. Deng (2003). Approximation and computation of arbitrage in frictional foreign exchange market (extended abstract). *Electronic Notes in Theoretical Computer Science* 78, 293–302.

J. Cheetham, F. Dehne, A. Rau-Chaplin, U. Stege, and P. J. Taillon (2003). Solving large FPT problems on coarse grained parallel machines. *Journal of Computer and System Sciences* 67(4), 691–706.

J. Chen, B. Chor, M. Fellows, X. Huang, D. W. Juedes, I. Kanj, and G. Xia (2005). Tight lower bounds for certain parameterized NP-hard problems. *Information and Computation* 201(2), 216–231.

J. Chen, X. Huang, I. Kanj, and G. Xia (2004). Linear FPT reductions and computational lower bounds. *STOC 2004*, 212–221.

J. Chen, I. A. Kanj, and G. Xia (2006). Improved parameterized upper bounds for Vertex Cover. *MFCS 2006*, 238–249.

J. Chen, Y. Liu, S. Lu, B. O'Sullivan, and I. Razgon (2008). A fixed-parameter algorithm for the directed feedback vertex set problem. *Journal of the ACM* 55(5).

J. Chen and J. Meng (2008). On parameterized intractability: hardness and completeness. *The Computer Journal* 51(1), 39–59.

B. Chor, M. R. Fellows, and D. W. Juedes (2004). Linear kernels in linear time, or how to save k colors in O(n) steps. *WG 2004*, 257–269.

The Computer Journal (2008). Two special issues of surveys of various aspects of parameterized complexity and algorithmics (Guest Editors: M. Fellows, R. Downey and M. Langston). *The Computer Journal,* Volume 51: Number 1 and Number 3.

B. Courcelle (1990) The Monadic Second-Order Logic of Graphs. I. Recognizable Sets of Finite Graphs *Inf. Comput*. 85(1): 12–75.

H. Dell and D. van Melkebeek (2010). Satisfiability allows no nontrivial sparsification unless the polynomial-time hierarchy collapses. *STOC 2010*, 251–260.

E. Demaine (2001). Playing games with algorithms: algorithmic combinatorial game theory. *MFCS 2001*, 18–32.

E. Demaine and M. T. Hajiaghayi (2008). The bidimensionality theory and its algorithmic applications. *The Computer Journal* 51(3), 292–302.

Discrete Optimization (2011). Special issue on parameterized complexity of discrete optimization (Guest Editors: M. R. Fellows, F. V. Fomin, and G. Gutin) 8(1).

R. G. Downey and M. R. Fellows (1995a). Fixed-parameter tractability and completeness I: basic results. *SIAM Journal on Computing* 24, 873–921.

R. G. Downey and M. R. Fellows (1995b). Fixed-parameter tractability and completeness II: on completeness for W[1]. *Theoretical Computer Science* 141, 109–131.

R. G. Downey and M. R. Fellows (1998). *Parameterized Complexity*. Springer-Verlag.

R. G. Downey, M. R. Fellows, B. Kapron, M. T. Hallett, and H. T. Wareham (1994). Parameterized complexity of some problems in logic and linguistics. *LFCS 1994*, 89–100.

R. G. Downey, M. R. Fellows, and C. McCartin (2006). Parameterized approximation algorithms. *IWPEC 2006*, 121–129.

R. G. Downey, M. Fellows, and U. Taylor (1997). The parameterized complexity of relational database queries and an improved characterization of W[1]. *DMTCS 1996*, 194–213.

M. Fellows (2002). Parameterized complexity: the main ideas and connections to practical computing. In: *Experimental Algorithmics*, Springer-Verlag, 51–77.

M. R. Fellows (2003). Blow-ups, win/win's and crown rules: some new directions in FPT. *WG 2003*, 1–12.

M. R. Fellows (2009). Towards fully multivariate algorithmics: some new results and directions in parameter ecology. *IWOCA 2009*, 2–10.

M. R. Fellows, F. V. Fomin, D. Lokshtanov, F. A. Rosamond, S. Saurabh, S. Szeider, and C. Thomassen (2011). On the complexity of some colorful problems parameterized by treewidth. *Information and Computation* 209(2), 143–153.

M. R. Fellows, D. Hermelin, F. A. Rosamond, and S. Vialette (2009a). *Theoretical Computer Science* 410(1), 53–61.

M. R. Fellows and N. Koblitz (1993). Fixed-parameter complexity and cryptography. *AAECC 1993*, 121–131.

M. R. Fellows and M. Langston (1987). Nonconstructive advances in polynomial time complexity. *Information Processing Letters* 26, 157–162.

M. Fellows, M. A. Langston. (1989a) On Search, Decision and the Efficiency of Polynomial-Time Algorithms (Extended Abstract) *STOC* 1989, 501–512.

M. Fellows, M.A. Langston(1989b) An Analogue of the Myhill-Nerode Theorem and Its Use in Computing Finite-Basis Characterizations (Extended Abstract) *FOCS* 1989: 520–525.

M. R. Fellows and F. Rosamond (2007). The complexity ecology of parameters: an illustration using bounded max leaf number. *CiE 2007*, 268–277.

M. R. Fellows, F. A. Rosamond, F. V. Fomin, D. Lokshtanov, S. Saurabh, and Y. Villanger (2009b). Local search: is brute-force avoidable? *IJCAI 2009*, 486–491.

H. Fernau, T. Hagerup, N. Nishimura, P. Ragde and K. Reinhardt (2003). On the parameterized complexity of the generalized rush hour puzzle. *CCCG 2003*, 6–9.

F. V. Fomin, D. Lokshtanov, V. Raman, and S. Saurabh (2010). Fast local search algorithm for Weighted Feedback Arc Set in Tournaments. *AAAI 2010*.

J. Flum and M. Grohe (2006). *Parameterized Complexity Theory*, Springer-Verlag.

S. Gaspers, D. Kratsch, and M. Liedloff. On independent sets and bicliques in graphs. *Algorithmica*, to appear.

S. Gaspers and S. Szeider (2011). Kernels for global constraints. *IJCAI 2011*, to appear.

G. Gottlob and S. Szeider (2008). Fixed-parameter algorithms for Artificial Intelligence, Constraint Satisfaction and Database Problems. *The Computer Journal* 51(3), 303–325.

M. Grohe (2002). The parameterized complexity of database queries. *PODS 2002*, 82–92.

J. Guo, H. Moser, and R. Niedermeier (2009). Iterative compression for exactly solving NP-hard minimization problems. *Algorithmics of Large and Complex Networks 2009*, 65–80.

J. Guo and R. Niedermeier (2007). Invitation to data reduction and problem kernelization. *SIGACT News*, March 2007, 31–45.

S. Hartung and R. Niedermeier (2010). Incremental list coloring of graphs, parameterized by conservation. *TAMC 2010*, 258–270.

S. Helwig, F. Hüffner, I. Rössling, and M. Weinard (2010). Selected design issues. *Algorithm Engineering 2010*, 58–126.

F. Henglein and H. G. Mairson (1991). The complexity of type inference for higher-order typed lambda calculi. *POPL 1991*, 119–130.

F. Hüffner (2009). Algorithm engineering for optimal graph bipartization. *Journal of Graph Algorithms and Applications* 13(2), 77–98.

F. Hüffner, R. Niedermeier, and S. Wernicke (2008). Techniques for practical fixed-parameter algorithms. *The Computer Journal*, 51(1):7–25.

Journal of Computer and System Sciences (2003). Parameterized computation and complexity. (Guest Editors: Jianer Chen, Michael R. Fellows), 67(4).

S. Khot and O. Regev (2008). Vertex cover might be hard to approximate to within $2 - \varepsilon$. *Journal of Computer and System Sciences* 74(3), 335–349.

C. Komusiewicz, R. Niedermeier, and J. Uhlmann (2009). Deconstructing intractability a case study for interval constrained coloring. *CPM 2009*, 207–220.

M. A. Langston, F. N. Abu-Khzam, R. L. Collins, M. R. Fellows, W. H. Suters, and C. T. Symons (2004). Kernelization algorithms for the Vertex Cover problem: theory and experiments. *ALENEX 2004*, 62–69.

M. A. Langston, F. N. Abu-Khzam, and P. Shanbhag (2003). Scalable parallel algorithms for difficult combinatorial problems: a case study in optimization. *PDCS 2003*, 649–654.

O. Lichtenstein and A. Pneuli (1985). Checking that finite-state concurrent programs satisfy their linear specification. *POPL 1985*, 97–107.

M. Mahajan and V. Raman (1999). Parameterizing above guaranteed values: MaxSat and MaxCut. *Journal of Algorithms* 31(2), 335–354.

D. Marx (2008a). Parameterized complexity and approximation algorithms. *The Computer Journal* 51(1), 60–78.

D. Marx (2008b). Searching the k-change neighborhood for TSP is W[1]-hard. *Operations Research Letters* 36(1), 31–36.

K. Mehlhorn (1984). *Data Structures and Efficient Algorithms, Volume 2: Graph Algorithms and NP-Completeness*, Springer.

N. Misra, V. Raman, and S. Saurabh (2011). Lower bounds on kernelization. *Discrete Optimization* 8(1), 110–128.

G. L. Nemhauser and L. E. Trotter (1975). Vertex packings: structural properties and algorithms. *Mathematical Programming* 8, 232–248.

R. Niedermeier (2006). *Invitation to Fixed Parameter Algorithms*. Oxford University Press.

R. Niedermeier (2010). Reflections on multivariate algorithmics and problem parameterization. *STACS 2010*, 17–32.

S. Ordyniak, D. Paulusma, and S. Szeider (2010). Satisfiability of acyclic and almost acyclic CNF formulas. *FSTTCS 2010*, 84–95.

S.-I. Oum (2005). Approximating rank-width and cliquewidth quickly. *WG 2005*, 49–58.

C. Papadimitriou and M. Yannakakis (1997). On the complexity of database queries. *POPL 1997*, 12–19.

R. Rizzi, P. Mahata, L. Mathieson, and P. Moscato (2010). Hierarchical clustering using the arithmetic-harmonic cut: complexity and experiments. *PLoS ONE* 5(12): e14067.

N. Robertson and P. D. Seymour (1983) Graph minors. I. Excluding a forest. J. Comb. Theory, Ser. B 35(1), 39–61.

M. Samer and S. Szeider (2010). Constraint satisfaction with bounded treewidth revisited. *J. Comput. Syst. Sci.* 76(2), 103–114.

A. Scott (2010). The parameterized complexity of finding short winning strategies in combinatorial games. Ph.D. dissertation, University of Victoria.

T. Shrot, Y. Aumann, and S. Kraus (2009). Easy and hard coalition resource game formation problems: a parameterized complexity analysis. *AAMAS (1) 2009*, 433–440.

C. Sloper and J. A. Telle (2008). An overview of techniques for designing parameterized algorithms. *The Computer Journal* 51(1), 122–136.

O. Suchy (2011). Parameterized complexity: nonstandard parameterizations of graph problems. Ph.D. dissertation, Charles University in Prague (Czech Republic).

S. Tazari, M. Müller-Hannemann (2009). Dealing with Large Hidden Constants: Engineering a Planar Steiner Tree PTAS. ALENEX 2009: 120–131.

I. van Rooij and T. Wareham (2008). Parameterized complexity in cognitive modeling: foundations, applications, and opportunities. *The Computer Journal* 51(3), 385–404.

T. Walsh (2010). Parameterized complexity results in symmetry breaking. In: Proceedings of the Fifth International Symposium of Parameterized and Exact Computation 2010 (Chennai, India), Springer, LNCS (6478) 4–13.

Chapter 14
Quantum Computing

Todd A. Brun

Introduction to Quantum Computing

One of the newest paradigms for a computing machine is the idea of a *quantum computer*: a computer that functions according to the laws of quantum mechanics that apply to the fundamental particles and forces of the world. Traditional computers are described by classical physics, which holds at ordinary human scales. Quantum effects are masked for such macroscopic systems. Indeed, so completely are these quantum effects hidden that their existence was not even suspected until the beginning of the twentieth century. And even at present, quantum mechanical behavior has only been produced reliably in very microscopic systems: single particles, atoms, and molecules.

To build a quantum computer requires an unprecedented ability to establish the state of a system, to isolate it from the environment, and precisely control its evolution. Only now is experimental physics approaching the level of precision needed. A quantum computer will require many quantum systems to be prepared and controlled jointly. The difficulty of this task raises the question: why should we make the effort? What advantage does a quantum computer have over an ordinary classical computer?

The answer is both surprising and exciting. Quantum computers can run fundamentally new kinds of algorithms—algorithms that draw on essentially quantum effects, such as superposition, entanglement, and interference. And it has been shown that certain problems have quantum algorithms that run faster than the best known classical algorithms—in some cases, provably faster than *any* classical algorithm. This promise has spurred a huge enterprise to both understand the theory of quantum computation, and to build experimental systems that are capable of carrying out quantum computations in practice.

T.A. Brun (✉)
University of Southern California, Los Angeles, CA, USA
e-mail: tbrun@usc.edu

E.K. Blum and A.V. Aho (eds.), *Computer Science: The Hardware, Software and Heart of It*, DOI 10.1007/978-1-4614-1168-0_14,

History of QM: Puzzles in the Classical World

Quantum Mechanics is now over 100 years old, and is one of the most successful scientific theories ever created. We believe it to be the underpinning of all physical laws. But at ordinary human scales, its effects are almost totally masked. Only by looking at phenomena at very short length and time scales can we see quantum behavior.

By the 1890s, *classical physics*—Newtonian mechanics plus Maxwell's electromagnetic theory and Boltzmann's statistical mechanics—seemed capable of explaining virtually all physical phenomena. But a number of seemingly minor puzzles proved to be gaps that would completely overthrow the classical structure of physics.

The first of these puzzles was the attempt by Max Planck to derive the proper distribution of thermal energy for an electromagnetic field. The model he used was a closed box heated at temperature T, empty except for whatever electromagnetic thermal radiation it contained. A standard classical derivation suggested that this "black-body" radiation should contain *infinite* energy. To get around this physical impossibility, in 1900, Planck came up with a derivation for a finite energy result. He assumed that the energy in each electromagnetic wave mode came in discrete chunks E proportional to the wave frequency f, that is, $E = hf$, with a constant of proportionality h (now called *Planck's constant*). This gave him a formula that exactly matched experiment. The constant of proportionality is the incredibly tiny $h = 6.6261 \times 10^{-34}\mathrm{kgm}^2/\mathrm{s}$; it is essentially because h is so small that quantum effects had never been seen.

Soon this constant began cropping up in many other physics puzzles: the photoelectric effect, explained by Albert Einstein in 1905 (and for which he later received the Nobel prize); the stability of atoms; and the discrete spectrum of atomic emissions, first roughly explained by Niels Bohr; the interference of "matter particles" that showed them sometimes to behave like waves. The solutions to all of these problems were found at first in an ad hoc manner, making changes to classical mechanics one at a time. Eventually the work of Schrödinger, Heisenberg, Dirac, von Neumann, Pauli, and others in the 1920s and 1930s swept classical physics away entirely for atomic and subatomic phenomena, replacing it with a new theory known as *quantum mechanics*. This is essentially the theory that we have today. We summarize it below.

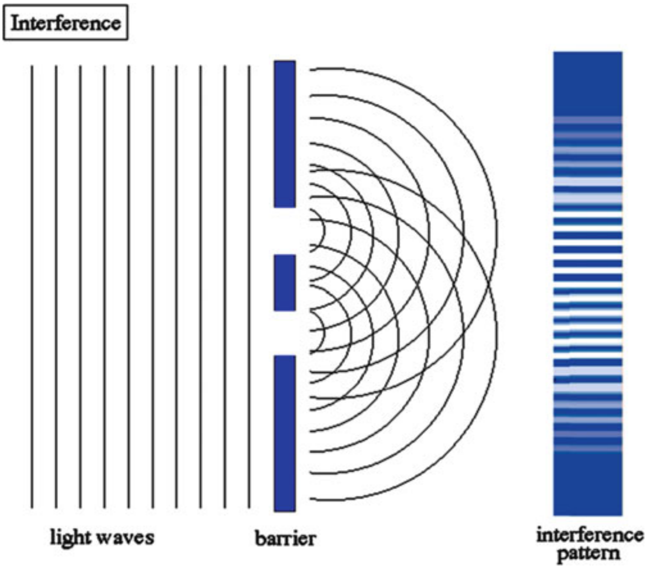
Properties of Quantum Mechanics

What are the revolutionary properties of quantum mechanics? Any quantum theorist can make his or her own list of distinctive quantum properties. Here is mine: *indeterminism, interference, uncertainty and complementarity, discrete spectra for bound systems, superposition (linearity)*, and *entanglement* (For comparison the reader may read about quantum mechanics in the book Mathematics of Physics and Engineering by Blum and Lototsky 2006). We will touch on several of these properties in this chapter, and come to understand a little bit of their technical meaning. But let us first get a more qualitative picture.

Indeterminism

The most fundamental distinction between classical and quantum mechanics is that classical mechanics is a *deterministic* theory: given perfect knowledge of the current state of a system, its properties at all past and future times is, in principle, determined precisely. In classical mechanics, probabilities only describe situations where one's knowledge is incomplete.

By contrast, quantum mechanics makes statements only about *probabilities* of properties. If the same measurement is performed on several identically prepared systems, one cannot in general expect the same outcome. This is not because we lack complete information about the systems described; rather, it is because the exact outcome of the measurement is inherently unpredictable (section "Interference"). Probabilities in quantum mechanics are not calculated directly, but from *probability amplitudes,*[1] which are complex numbers associated with state functions. The probability of a state is the *square* of the probability amplitude. Their relationship is similar to that between the amplitude of a wave and its intensity (which is the square).



[1] The probability interpretation of QM was first suggested by Max Born. It was in a footnote to this pioneering paper, added in proof, that he realized that probabilities were not equal to amplitudes, but to their squares

Interference

For example, in the two-slit experiment diagrammed here, the probability amplitude for a particle to hit a particular point on the screen is the sum of the amplitude to go through slit A and hit the point, and the amplitude to go through slit B and hit the point. However, the probability to hit the point is then $p = |\alpha_A + \alpha_B|^2$. Because the amplitudes can either add or cancel each other out, this system exhibits *interference fringes*. Some parts of the screen will not be hit by particles, even though from each slit there is a nonzero amplitude to reach that part of the screen. Other parts will be hit at a higher rate.

Uncertainty

For a classical particle, complete information is given by the position x of the particle and its velocity (or momentum) p. For a quantum particle, this is not the case. As famously realized by Heisenberg, a measurement of a particle's position disturbs its momentum, with the size of the disturbance proportional to the precision of the measurement. Similarly, a measurement of the momentum disturbs the particle's position. This constraint on the precision with which position and momentum can be measured simultaneously is quantified by the inequality

$$(\Delta x)^2(\Delta p)^2 \geq \frac{\hbar^2}{4}, \hbar = h/2\pi.$$

This constraint could be seen as just a practical limitation on the precision of measurements; we might imagine that particles really do have precise positions and momenta, which we are unable to exactly determine. However, in quantum mechanics an even more radical explanation holds: *in fact, the two quantities are not even simultaneously well-defined.*

Complementarity

This idea—that different ways of describing a system may be mutually exclusive—is called *complementarity*. For position and momentum, this means that we can write down amplitudes for every possible position of a particle, OR for every possible momentum; but not both, because those quantities cannot be simultaneously measured. If $\Psi(x)$ is the *wavefunction* giving the probability amplitude to be at every point x, we can also write down $\tilde{\Psi}(p)$(the Fourier transform) which gives the amplitude for every p. But there is no similar function $\Psi(x,p)$. For variables other than position and momentum, similar statements hold. In particular, for the kind of *discrete* variables used in quantum information theory, different kinds of uncertainty and complementarity apply.

Discrete Spectrum

For the Bohr atom, only certain discrete orbits are allowed, with discrete values of the energy. These values are called *energy levels*. This pattern of discrete spectra is common to bound systems in quantum mechanics.

This discreteness is useful in quantum information theory, because it matches the discreteness assumed in quantifying classical information. For instance, the simplest quantum system would be one with only two distinct levels. This is analogous to a classical bit, which can take one of two possible values. In quantum information theory, most of the systems we will deal with have only a finite number of discrete levels. However, while the number of energy levels may be finite, the possible states are *continuous*. This is because of another property of quantum mechanics: *linearity*.

Superposition

Suppose that Ψ and ϕ are two valid "states" of a quantum system (that is, two possible *wavefunctions*.) Then $a\Psi + b\phi$, where a and b are complex numbers, is *also* a valid state of the system. This is an example of a *superposition*.

The reason such superpositions are possible is because quantum mechanics is a *linear* theory: the set of all states forms a complex vector space (or, more precisely, a Hilbert space). The *evolution equation* for states, the *Schrödinger equation*, is a linear differential equation. The various physical operations on quantum mechanical systems can be represented by linear operators (matrices) on the Hilbert space. Various physical interpretations can be given to superposition. One famously paradoxical situation has to do with a cat's state. It is linearity that makes possible the famous Schrödinger's Cat paradox, in which a cat can be both alive (Ψ) and dead (ϕ). Strictly speaking, the cat is in a superposition of alive and dead.

Entanglement

This last property of quantum mechanics is one of the most difficult to explain; but it plays a crucial role in quantum computation and quantum information. If a quantum system consists of multiple subsystems—for instance, of several distinct particles—it is possible for the *joint* system to have a definite state Ψ, while *none* of the subsystems has a well-defined state. In this situation, the subsystems are said to be *entangled*. We can have for two subsystems x and y,

$$\Psi(x,y) \neq \Psi(x)\Psi(y).$$

While this may sound like an unusual and exotic situation, in fact it is not. Almost all states of multiple subsystems are entangled. But the effects of entanglement are masked at larger scales. At the quantum level, entanglement behaves very much like classical correlation: the outcomes of measurements on different subsystems are correlated. But these correlations can be stronger than any possible classical correlation. This result was proven by John Bell in the 1960s, and experimentally demonstrated by Clauser and by Aspect in the 1970s and 1980s.

Much has been made of entanglement, including various assertions that quantum mechanics is *nonlocal*: that once in contact, quantum systems continue to influence each other even when far apart. These assertions are a bit overstated. But it is true that entanglement is qualitatively different from any phenomenon that occurs in classical physics.

Quantum Information and Computation—A Prehistory

As microelectronic components get smaller and smaller, computer chips are steadily approaching the point where quantum effects must be taken into account (see Chap. 5 appendix). However, by the 1980s, some people were already starting to ask if quantum mechanics could actually be exploited to make new information processing techniques possible.

The first to propose an intrinsically quantum mechanical computer was P. Benioff in 1980. Y. Manin and R. Feynman both proposed that a quantum computer might be able to efficiently simulate quantum systems—something that ordinary classical computers find very difficult (Feynman 1982; Benioff 1982; Manin 1980). This idea of quantum simulation remains one of the most important potential applications of a quantum computer.

But is a quantum computer even possible? As we shall see, such a computer must operate *reversibly*; that is, it cannot dissipate energy. Ordinary computers are highly dissipative, as anyone who has ever felt the heat they give off has noticed (see Chap. 8).

In the 1970s, Charles Bennett of IBM showed that any computation can, in principle, be done reversibly, building on work from the 1960s by Rolf Landauer. That is, in principle, there is no requirement that a computer consume power to operate (though it may take energy to start a computation). This paved the way for the possibility of reversible quantum computers.

In 1985, David Deutsch presented a new idea. Because of superposition, one can imagine a quantum computer in a superposition of *different computations*. For instance, a computer could simultaneously calculate the value of a function $f(x)$ for every possible input x in a single run. Deutsch called this possibility *quantum parallelism*, and speculated that, just like ordinary parallelism, it would increase the computing power.

Naive applications of quantum parallelism add nothing to the power of the computer. But in 1988, Deutsch found a clever algorithm that indirectly exploited quantum parallelism to solve a problem more efficiently than any possible classical computer. This problem was rather artificial, but it was the first example where a quantum computer could be shown to be more powerful than a classical computer (Deutsch 1985; 1989).

Meanwhile, in 1984, Charles Bennett and Gilles Brassard found another way in which quantum properties could be exploited for information processing. They exploited the uncertainty principle as a way to distribute a cryptographic key with perfect security (see Chap. 11). Single quanta are used to send the bits of the key, in one of two possible complementary variables. If an eavesdropper tries to intercept the bits and measure them, it disturbs them in such a way that it can always be detected. This and similar schemes are called *quantum cryptography*, and it is the first quantum information protocol that has been translated into a real technology (Bennett and Brassard 1984).

Artur Ekert, in 1991, proposed another scheme for quantum cryptography, this one based on entanglement rather than uncertainty. Bennett and collaborators found other uses for entanglement: *quantum teleportation* in which separated experimenters sharing two halves of an entangled system can make use of the entanglement to transfer a quantum state from one to the other using only classical communication; and *superdense coding*, in which sending a single quantum bit allows the transmission of two bits of classical information.

Richard Josza and David Deutsch extended Deutsch's original algorithm to a more general, but still artificial version of the same problem; and D.R. Simon found another problem, albeit still rather specialized, in which quantum computers outperformed classical computers. The stage was being set for the real breakthrough.

In 1994, Peter Shor of AT&T published a paper showing that a quantum computer could decompose a large number into its prime factors in a time of polynomial order in the length of the number. The difficulty of factoring by classical computers accounts for success of the RSA public-key encryption algorithms, which are the basis for many secure transactions on the world wide web (see Chap. 11). Suddenly, it was known that quantum computers could in principle solve a problem of importance in the real world (Shor 1994). This was followed in 1996 by Grover's unstructured search algorithm, which gave the first provably better result by a quantum computer over a classical computer (Grover 1996).

Rather than being an obscure interest for a handful of physicists and computer scientists, quantum information processing was suddenly of interest to researchers in many fields. In the years since the factoring algorithm was discovered, the field of quantum information theory has exploded, and so have the experimental efforts to realize quantum computation in practice.

The Mathematical Structure of Quantum Theory

The Stern-Gerlach Experiment and Spin

The most fundamentally useful system in quantum computing, and its physical and mathematical properties, are most easily illustrated by the simplest quantum system ever discovered: the spin-1/2 particle. Experiments in the early 1920s discovered a new aspect of nature, and at the same time found the simplest quantum system in existence. In the Stern-Gerlach experiment, a beam of hot atoms is passed through a nonuniform magnetic field. This field exerts a force on the magnetic dipole moment of the atom, if any, and deflects it.

This experiment discovered two surprising things. First, the atoms do have a magnetic dipole moment. (Actually, the experiment saw the magnetic dipole moment of the *electron*.) In effect, in addition to being charged, electrons act like tiny bar magnets. They also, as it developed, have a tiny intrinsic amount of angular momentum, equal to $\hbar/2$. This quantity is called *spin*, and all known elementary particles have nonzero spin. Electrons are *spin-1/2* particles.

The second surprising thing was *how much* the path of the electrons was deflected. If electrons were classical bar magnets, they could be oriented in any direction. The component which was oriented the same way as the magnetic field gradient (say the Z direction) would determine the force on the electron, and hence how much they would be deflected. If electrons were like ordinary magnets with random orientations, they would show a continuous distribution of paths. The photographic plate in the Stern-Gerlach experiment would have shown a continuous distribution of impact positions.

What was observed was quite different. The electrons were deflected either up or down by a constant amount, in roughly equal numbers. Apparently, the Z component of the electron's spin is *quantized*: it can take only one of two discrete values. We say that the spin is either *up* or *down* in the Z direction. This is an embodiment of a *two-level system* or *quantum bit*, commonly called a *qubit*.

Sequential Measurements

Suppose an electron is passed through a Stern-Gerlach device, and is found to have spin up in the Z direction. If we pass it through a *second* Stern-Gerlach device, it will always be found to still have spin up in the Z direction. So this seems like an actual property of the spin, which we are measuring with the Stern-Gerlach device.

In a similar way, we can tilt the Stern-Gerlach apparatus 90° on its side and measure the component of spin in the X direction. Here again, that component of the spin is discrete: it is either up or down in the X direction. In fact, we can rotate it by any angle θ that we like, and measure the component of the spin in *any* direction; and it will always be found to have a discrete value, up or down, in that direction.

Suppose that we have determined the spin to be up in the Z direction, and we pass the spin through a second device to measure the X component of the spin. In this case, we get spin up or down in the X direction with *equal* probabilities. If we start with Z down, the same thing happens. Suppose now that we measure Z up and then X up. What happens if we measure Z again?

In this case, we get Z up or down with equal probability! Measuring X has erased our original value of Z. Similarly, if we started with a definite state of X and measure Z, we erase the original value of X.

By a slightly more complicated arrangement, we can also measure the component of spin in the direction Y. If we do this, we find that measuring Y erases the original value of either X or Z; measuring X erases either Y or Z; and measuring Z erases either X or Y.

Complementarity and Randomness

The X, Y, and Z components of the electron spin are all *complementary variables*. Knowing one of the three precludes knowing the other two. They are not all simultaneously well-defined. If a given variable is *not* well-defined for a given state of the system, when we measure it the outcome is random.

Suppose that a spin is up in the Z direction. If we measure the component of spin along an axis at angle θ to the Z axis, we find the spin up along that axis with probability $p_{up} = \cos^2(\theta/2)$, and spin down along that axis with probability $p_{down} = \sin^2(\theta/2)$. (This result is not special to the Z direction. If a spin is up (or down) along any axis, measuring along another axis at an angle θ has outcomes with probabilities $\sin^2(\theta/2)$ and $\cos^2(\theta/2)$.) We want to find a mathematical framework to encompass all these results.

The Mathematical Description of Spin

To describe the state of a spin-1/2, we first choose a particular measurement to serve as a reference. By convention, this is usually the component of spin in the Z direction. Starting from this, there are two special states: spin definitely up or definitely down in the Z direction. We will write these states as $|\uparrow\rangle$ and $|\downarrow\rangle$. (This notation, in which the state is written in angled brackets, was introduced by Paul Dirac, and is commonly used in the field of quantum information and computation.)

Most states, however, do not give a definite outcome for a Z measurement. Instead, if we measure a spin in a general state, we will find it up or down in the Z direction with some probability. It turns out that we can represent these general states by a linear combination of our two special states:

$$|\Psi\rangle = \alpha|\uparrow\rangle + \beta|\downarrow\rangle.$$

In other words, a general state (or *wavefunction*) Ψ is a linear combination of these special states, which serve as a *basis* for the space of all states. This set of all states forms a vector space. The values α and β are the *probability amplitudes* in the up and down directions; and if we measure the *Z* component of spin, we get up or down with probabilities equal to the squares of the amplitudes, $|\alpha|^2$ and $|\beta|^2$. Therefore, as probabilities, we require that $|\alpha|^2 + |\beta|^2 = 1$, or in other words, that the state be *normalized.*

Of course, there is nothing fundamental about our choice of measurement. We could, for example, have chosen to measure the component of spin in the *X* direction. In that case, there would again be two special states $|\uparrow_X\rangle$ and $|\downarrow_X\rangle$ that when measured always give the result up or down along the *X* direction. These states also form a basis for the space of all states. What do$|\uparrow_X\rangle$ and $|\downarrow_X\rangle$ look like in terms of our original basis states (representing the spin component in the *Z* direction)? It turns out that if the spin is up or down in the *X* direction, at an angle $\theta = \pi/2$ to the *Z* axis, we can write the states as

$$|\uparrow_X\rangle = (|\uparrow\rangle + |\downarrow\rangle)/\sqrt{2},$$
$$|\downarrow_X\rangle = (|\uparrow\rangle - |\downarrow\rangle)/\sqrt{2},$$

That works fine for the *X* direction. But what about the *Y* direction? That is also at an angle $\pi/2$ to the *Z* axis. But $|\uparrow_Y\rangle$ and $|\downarrow_Y\rangle$ can't be the same as $|\uparrow_X\rangle$ and $|\downarrow_X\rangle$. In this case, we solve the problem by letting the amplitudes be *complex numbers.* So, with the imaginary unit *i*,

$$|\uparrow_Y\rangle = (|\uparrow\rangle + i|\downarrow\rangle)/\sqrt{2},$$
$$|\downarrow_Y\rangle = (|\uparrow\rangle - i|\downarrow\rangle)/\sqrt{2},$$

We can see that these states produce the probabilities seen in the Stern-Gerlach experiment. If we think of $|\uparrow\rangle$ and $|\downarrow\rangle$ as being basis vectors for a two-dimensional complex vector space, then in terms of this basis, we can write any state as a column vector:

$$|\Psi\rangle = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}.$$

We will make the assumption that $|\uparrow\rangle$ and $|\downarrow\rangle$ are orthogonal vectors of unit length. By making this assumption, the basis we have chosen is orthonormal, and a state is normalized if it forms a unit vector under the usual complex inner product.

The Effect of Measurement

As we saw above, if we measure a spin in the state $|\Psi\rangle = \alpha|\uparrow\rangle + \beta|\downarrow\rangle$, we get result "up" with probability $|\alpha|^2$ and "down" with probability $|\beta|^2$. But the state does not remain the same after such a measurement. Rather, if the result is "up" then the state of the spin after measurement is $|\uparrow\rangle$ and if the result is "down" the state is $|\downarrow\rangle$. This means that if we repeat the same measurement immediately, we will get the same result.

This *also* means that the act of measurement disturbs the state. Suppose you are given a spin that has been prepared in an unknown (to you) state $|\Psi\rangle$. If you measure that spin in the Z direction, you will get either "up" or "down." But that will not be sufficient to tell you what the state $|\Psi\rangle$ was *before* the measurement, and no further measurements will reveal any more information. Measurement causes an irreversible change in the state.

This also means that a spin can have a well-defined component in a single direction, but it cannot have a well-defined component in more than one direction at a time. Measuring spin in a new direction disrupts its value in the old direction. This is the phenomenon of complementarity discussed above, and is related to the well-known uncertainty principle of Heisenberg.

Global Phase

Suppose that we multiply the state $|\Psi\rangle$ by a *pure phase* $\exp(i\phi)$,

$$\alpha \rightarrow e^{i\phi}\alpha, \quad \beta \rightarrow e^{i\phi}\beta.$$

This doesn't change the probabilities for a measurement along the Z axis. Nor does it change the probabilities for a measurement along *any other* axis.

This means that multiplying by a pure phase has *no observable physical consequences*. We say that the *global* phase of a state is *arbitrary*. (If we multiplied α and β by *different* phases that *would* have observable consequences. It still wouldn't change the probabilities for a Z measurement, but it would change the probabilities for other measurements.)

If we fix the normalization $|\alpha|^2 + |\beta|^2 = 1$ and the global phase (so, for instance, α is real), then there are only two independent parameters for the state of a spin. One useful choice of parameters is

$$|\Psi\rangle = \cos(\theta/2)|\uparrow\rangle + e^{i\phi}\sin(\theta/2)|\downarrow\rangle,$$

where $0 \leq \theta \leq \pi$ and $-\pi \leq \phi \leq \pi$. These are the coordinates of points on the surface of a sphere. This is called the *Bloch Sphere Representation*. Each point on the sphere corresponds to a state; states corresponding to opposite points are orthogonal. $|\uparrow\rangle$ and $|\downarrow\rangle$ are the north and south poles at $\theta = 0, \pi$.

Evolution of the State

A nonuniform magnetic field produces a net force on a particle with spin. This is the basis of the Stern-Gerlach apparatus. A uniform magnetic field does not produce a net force. But it does cause the direction of the spin to rotate, or *precess*.

We describe this effect mathematically using a set of 2 × 2 matrices called the *Pauli matrices*:

$$\hat{X} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \hat{Y} = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \hat{Z} = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}.$$

These matrices act on the two-dimensional vector space that is the set of spin-1/2 states.

The time-evolution of the state is given by the Schrödinger equation

$$ih \frac{d|\Psi\rangle}{dt} = \hat{H}|\Psi\rangle,$$

where $\hat{H} = \hat{H}^\dagger$ is an Hermitian operator known as the *Hamiltonian*, and which describes the *energy* of the system. For a spin-1/2 in a uniform magnetic field in the direction $\vec{n} = (n_x, n_y, n_z)$ (where $\vec{n}$ is a unit vector in space), the Hamiltonian is

$$\hat{H} = E(n_x\hat{X} + n_y\hat{Y} + n_z\hat{Z}),$$

where E is an energy proportional to the strength of the magnetic field. If the spin is initially in the state $|\Psi(t_1)\rangle$ at time t_1, and evolves until time $t_2 > t_1$, then the state is transformed by a *unitary transformation* $\hat{U}(t_2, t_1)$. A unitary matrix satisfies $\hat{U}^\dagger\hat{U} = \hat{U}\hat{U}^\dagger = \hat{I}$, where $\hat{U}^\dagger$ is the Hermitian conjugate of $\hat{U}$—that is, the complex conjugate of the transpose.

This is easiest to see if $\hat{H}$ is a fixed operator (i.e., constant in time). In that case, a solution to Schrödinger's equation is

$$|\Psi(t_2)\rangle = \exp(-i\hat{H}(t_2 - t_1)/\hbar)|\Psi(t_1)\rangle.$$

The operator $-\hat{H}(t_2 - t_1)/\hbar$ is Hermitian, so the operator

$$\hat{U}(t_2, t_1) = \exp(-i\hat{H}(t_2 - t_1)/\hbar)$$

is unitary.

Suppose there is a uniform magnetic field in the Z direction. Then states with spin up and down along the Z axis have different energies. This is represented by a Hamiltonian

$$\hat{H} = \begin{pmatrix} E_0 & 0 \\ 0 & -E_0 \end{pmatrix} \equiv E_0\hat{Z},$$

where E_0 is proportional to the strength of the magnetic field. If $|\Psi\rangle = \alpha|\uparrow_Z\rangle + \beta|\downarrow_Z\rangle$ at $t = 0$, then

$$|\Psi(t)\rangle = \alpha e^{-iE_0t/\hbar}|\uparrow_Z\rangle + \beta e^{iE_0t/\hbar}|\downarrow_Z\rangle.$$

This type of evolution is equivalent to a steady rotation about the Z axis, called *precession*. If $|\Psi\rangle$ is an eigenstate of $\hat{H}$, $|\uparrow_Z\rangle$ or $|\downarrow_Z\rangle$, the only effect is a change in the global phase of the state, which has no physical consequences. Because of this, we call these *stationary states*.

Similarly, we could have a uniform field in the X direction or the Y direction. In these cases, the Hamiltonians would be $E_0\hat{X}$ or $E_0\hat{Y}$, and the stationary states would be the $\hat{X}$ or $\hat{Y}$eigenstates.

If the Hamiltonian is not constant, $\hat{H}=\hat{H}(t)$, then the situation is more complicated; but the time evolution in every case is still given by a unitary transformation. A common situation in quantum information is when we have some control over the Hamiltonian of the system. For instance, we could turn on a uniform magnetic field in the Z direction, leave it on for a time τ, and then turn it off. In that case, the state will have evolved by

$$|\Psi\rangle \rightarrow \exp(i\theta\hat{Z})|\Psi\rangle \equiv \hat{U}|\Psi\rangle,$$

where $\theta = -E_0\tau/\hbar$. In this case, we say we have "performed a unitary transformation $\hat{U}$ on the system." The Hamiltonian has the time dependence $\hat{H}(t) = f(t)E_0\hat{Z}$ where

$$f(t) = \begin{cases} 0, & \text{if } t<0; \\ 1, & \text{if } 0 \le t \le \tau; \\ 0 & \text{if } t>\tau. \end{cases}$$

One final point. Suppose that we included in our Hamiltonian a term proportional to the identity matrix: $E_0\hat{I}$. What effect would this have on the evolution of the state? In fact, its only effect would be to multiply the state by a global phase, which as we have seen above has no physical meaning. Because of this, we will always allow ourselves the freedom to add or subtract such a term from any Hamiltonian if we like.

Systems of More than One Spin

The spin-1/2 is the simplest quantum system in existence, which is its virtue. But at the same time, it is inadequate to describe more complicated systems. Similarly, a computer with only a single bit of memory is not capable of very much computation. So let us now see how to describe systems consisting of *multiple* spins.

For a single spin, we first chose a canonical measurement (the Z component of spin) and identified two special states $|\uparrow\rangle$ and $|\downarrow\rangle$ that each gave the result up and down, respectively, with probability 1. These two states then served as a basis for general states. We will follow the same procedure with n spins. Our canonical measurement is to measure the Z component of each spin individually; there are therefore 2^n possible results, which give us our basis states: $|\uparrow\cdots\uparrow\uparrow\rangle, |\uparrow\cdots\uparrow\downarrow\rangle, ..., |\downarrow\cdots\downarrow\downarrow\rangle$, and the most general state is a linear combination of all of these. In the simplest case, a state for two spin-1/2 systems has four components:

$$|\Psi\rangle = \alpha_{\uparrow\uparrow}|\uparrow\uparrow\rangle + \alpha_{\uparrow\downarrow}|\uparrow\downarrow\rangle + \alpha_{\downarrow\uparrow}|\downarrow\uparrow\rangle + \alpha_{\downarrow\downarrow}|\downarrow\downarrow\rangle.$$

It is possible that each of the two spins has its own quantum state $\alpha_{1,2}|\uparrow\rangle + \beta_{1,2}|\downarrow\rangle$, in which case their joint state will be a product of these two states:

$$\alpha_{\uparrow\uparrow} = \alpha_1\alpha_2, \alpha_{\uparrow\downarrow} = \alpha_1\beta_2, \alpha_{\downarrow\uparrow} = \beta_1\alpha_2, \alpha_{\downarrow\downarrow} = \beta_1\beta_2.$$

We have used a simple juxtaposition notation in the Dirac bracket to denote "product" states. The strictly rigorous mathematical structure of a product state is given by the tensor product of the factors. (See the Blum and Lototsky book.) Here we use the informal Dirac notation which has a natural interpretation as a product state. But generally, most states $|\Psi\rangle$ of a composite system are *not* product states. For an example with two spin-1/2 systems,

$$|\Psi\rangle = \frac{1}{\sqrt{2}}|\uparrow\downarrow\rangle - \frac{1}{\sqrt{2}}|\downarrow\uparrow\rangle$$

is not a product state. For this joint state, we cannot assign well-defined states to the subsystems. Such a joint state is called *entangled*. One consequence of entanglement is that measurements on the subsystems will generally be *correlated*.

For the purposes of this chapter we need to understand two other properties of multi-spin states. If we measure all n spins in the canonical measurement, the probability of a particular outcome (say $\uparrow\cdots\uparrow$) is the absolute value of the probability amplitude squared (in this case, $|\alpha_{\uparrow\ldots\uparrow}|^2$). After this measurement, the system will be left in the corresponding basis state.

But what if, instead of measuring all the spins, we measure only a *single* spin? Without loss of generality, let us suppose that we measure the first spin. In that case, we collect together all the terms where that spin is up, and all the terms where that spin is down, and write the whole state in this form:

$$|\Psi\rangle = \alpha|\uparrow\rangle|\Psi_\uparrow\rangle + \beta|\downarrow\rangle|\Psi_\downarrow\rangle,$$

where $|\Psi_\uparrow\rangle$ and $|\Psi_\downarrow\rangle$ are normalized states for the remaining $n-1$ spins. In this case, the probability of getting spin up or down for the first spin is $|\alpha|^2$ or $|\beta|^2$, respectively, and the system is left either in the state $|\uparrow\rangle|\Psi_\uparrow\rangle$ or the state $|\downarrow\rangle|\Psi_\downarrow\rangle$.

Next, how do states of n spins evolve with time? In the case of a single spin-1/2, we saw that time evolution was described by a 2×2 unitary matrix, which arises from the Schrödinger equation with a particular choice of Hamiltonian. With n spins, the states are 2^n-dimensional, and the most general time evolution is described by a $2^n \times 2^n$ unitary matrix.

Of course, in practice we are limited in the kinds of time evolution we can produce by the nature of the physical systems we use. So instead let us consider the special case where the evolution affects only one or two spins at a time. A simple example would be to apply a magnetic field to one spin, but not the others. We treat this in the same way that we treated a measurement of a single spin. We again write $|\Psi\rangle = \alpha|\uparrow\rangle|\Psi_\uparrow\rangle + \beta|\downarrow\rangle|\Psi_\downarrow\rangle$, and then apply our 2 × 2 unitary matrix just to the first spin. If the 2 × 2 unitary matrix $\hat{U}$ is

$$\hat{U} = \begin{pmatrix} a & b \\ c & d \end{pmatrix},$$

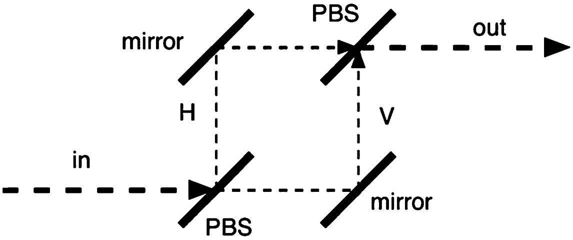
then after the transformation the state becomes $|\uparrow\rangle(a\alpha|\Psi_\uparrow\rangle + b\beta|\Psi_\downarrow\rangle) + |\downarrow\rangle(c\alpha|\Psi_\uparrow\rangle + d\beta|\Psi_\downarrow\rangle)$. In a similar way, one can apply a 4 × 4 unitary matrix to two of the spins, while leaving the others unchanged. We will see later that it is possible to build up any unitary matrix by using only single-spin and two-spin unitaries in succession.

Other Two-Level Systems

While the spin-1/2 is extremely simple, we can find other physical systems which behave in the same way. The most obvious example is the *polarization of a photon*. If light shines on a polarizing beam splitter (PBS), light with opposite polarizations (say H and V) exits from the two ports. If a single photon arrives at a PBS, it exits from one of the two ports *with some probability*. So a polarizing beam splitter for a photon acts just like a Stern-Gerlach apparatus for spin-1/2!

In fact, the mathematical description of spin-1/2 maps directly onto the case of photon polarization. The states $|\uparrow\rangle$ and $|\downarrow\rangle$ become the linear polarization states $|\mathbf{H}\rangle$ and $|\mathbf{V}\rangle$. $|\uparrow_X\rangle$ and $|\downarrow_X\rangle$ become the linear polarizations at 45° to H and V. And $|\uparrow_y\rangle$ and $|\downarrow_y\rangle$ become the *circular* polarizations $(|\mathbf{H}\rangle \pm i|\mathbf{V}\rangle)/\sqrt{2}$.

Note that having split the two components H and V, we can rejoin them to reconstruct the original state. (In principle, we can do this with the Stern-Gerlach apparatus as well; this is called *matter interferometry*.) So we see that "measurements" are not necessarily final until the actual *read-out* process is complete. Bohr called this final step an "irreversible classical amplification."



Just like Stern-Gerlach devices, we can construct PBSs to measure any polarization. The probabilities for different outcomes obey exactly the same mathematics as the probabilities for spin-1/2. Other systems can act like spin-1/2 as well: the two energy levels of the hyperfine splitting, the presence or absence of a photon in a cavity, etc. In these cases, we are really picking out a *two-dimensional subspace* of a larger space. All such two-dimensional systems are examples of *quantum bits*, or *qubits*. We can consider them to be fundamental building blocks of quantum information, just as we can consider ordinary bits to be fundamental building blocks of classical information.

Because of this universal mathematical structure, from here on we will no longer assume a particular physical embodiment of our qubits or of our quantum computer. Instead of spins up or down along a particular direction, we will choose a particular basis corresponding to some canonical single-qubit measurement. We label the basis states $|0\rangle$ and $|1\rangle$, where we make contact with the description of the spin-1/2 system by identifying

$$|0\rangle \equiv |\uparrow\rangle, |1\rangle \equiv |\downarrow\rangle.$$

We call this basis "the computational basis" or "the standard basis" or (by analogy to the spin-1/2 case) "the Z basis." For most the rest of this chapter, we will work in terms of this description. The only exception is near the end, when we will briefly discuss the current experimental state of the art in implementing quantum computation, and the physical systems used.

Quantum Information Processing

From this mathematical description we see what elements we can use to build information processing protocols.

1. We can prepare quantum systems in particular states.
2. We can perform a series of unitary transformations and measurements on the systems, where later operations can be conditioned on the results of earlier measurements.
3. The output of the process must be the result of some final measurement or measurements.

We will see in the remainder of this chapter how, from these building blocks, we can construct information processing protocols more powerful than any that can be run on a classical computer.

For a concise exposition on the mathematics of quantum mechanics and quantum computing, see Blum and Lototsky (2006). For a longer, more comprehensive treatment, the best reference remains Nielsen and Chuang (2000). The latter also includes an extensive list of references.

General Unitary Transformations

One-Bit Unitaries and Bloch Sphere Rotation

The most general spin-1/2 (2 × 2) Hamiltonian is:

$$\hat{H} = b\hat{X} + c\hat{Y} + d\hat{Z} \equiv E_0\vec{n}\cdot\vec{\sigma},$$

for $E_0 = \sqrt{b^2 + c^2 + d^2}$, where $\vec{n} = (n_x, n_y, n_z) = (b/E_0, c/E_0, d/E_0)$, with $n_x^2 + n_y^2 + n_z^2 = 1$ and $\vec{\sigma} = (\hat{X}, \hat{Y}, \hat{Z})$.

This produces a unitary operator

$$\exp\left(-i\hat{H}t/\hbar\right) = \cos(E_0 t/\hbar)\hat{I} - \mathrm{i}\sin(E_0 t/\hbar)\vec{n}\cdot\vec{\sigma}.$$

In the Bloch sphere picture, this evolution corresponds to a rotation around the axis $\vec{n}$ at a rate $E_0/\hbar$.

Now suppose that we can turn the Hamiltonian on and off. By turning a Hamiltonian on for a particular length of time, we can "rotate" the state by a particular angle. For a spin-1/2, this means we perform the unitary transformation

$$\hat{U}(\theta) = \cos(\theta/2)\hat{I} - i\sin(\theta/2)\vec{n}\cdot\vec{\sigma}.$$

Building up Unitaries

The important thing to remember is that any *product* of unitary operators is *also* unitary:

$$\hat{U}^\dagger\hat{U} = \hat{V}^\dagger\hat{V} = \hat{I}.$$
$$(\hat{U}\hat{V})^\dagger(\hat{U}\hat{V}) = \hat{V}^\dagger\hat{U}^\dagger\hat{U}\hat{V} = \hat{I}.$$

Suppose there are two *different* Hamiltonians we can turn on: $\hat{H}_1$ and $\hat{H}_2$. Then we can perform the unitaries

$$\hat{U}_1(\tau) = \exp\left(-i\hat{H}_1\tau/\hbar\right),$$
$$\hat{U}_2(\tau) = \exp\left(-i\hat{H}_2\tau/\hbar\right).$$

But we can *also* do the unitaries $\hat{U}_2(\tau_2)\hat{U}_1(\tau_1)$ and $\hat{U}_2(\tau_3)\hat{U}_1(\tau_2)\hat{U}_2(\tau_1)$ and $\hat{U}_2(\tau_n)\hat{U}_1(\tau_{n-1})\cdots\hat{U}_2(\tau_2)\hat{U}_1(\tau_1)$. Let's see how this works for the spin-1/2.

Suppose we can turn on Hamiltonians

$$\hat{H}_1 = E_x\hat{X},\ \hat{H}_2 = E_y\hat{Y}.$$

These produce unitaries $\hat{U}_1(\theta)$ and $\hat{U}_2(\theta)$, which correspond, in the Bloch sphere representation, to rotations by θ about the X and Y axes, respectively. There is a theorem in geometry that a rotation by any angle θ around any axis $\vec{n}$ can be done by doing three rotations in a row:

$$R_{\vec{n}}(\theta) = R_X(\phi_3)R_Y(\phi_2)R_X(\phi_1)$$

for some ϕ_1, ϕ_2, ϕ_3. Since every 2×2 unitary is equivalent to a Bloch sphere rotation about some axis $\vec{n}$, any 2×2 unitary equals

$$\hat{U}_1(\tau_3)\hat{U}_2(\tau_2)\hat{U}_1(\tau_1)$$

for some $\tau_1,\ \tau_2,\ \tau_3$ (up to an overall phase).

Two-Qubit unitaries

Unitaries that act on two qubits at a time represent some kind of *interaction* between them. Here is an example for two spin-1/2s:

$$\hat{H}_{\text{int}} = E_{\text{int}}\hat{Z}_1\hat{Z}_2,$$

where $\hat{Z}_1$ acts on the first spin and $\hat{Z}_2$ on the second spin. After a time τ this gives rise to a unitary transformation

$$\hat{U}(\theta) = \cos(\theta/2)\hat{I} - i\sin(\theta/2)\hat{Z}_1\hat{Z}_2.$$

where $\theta = 2E\tau/\hbar$. Suppose we have an initial product state

$$|\Psi\rangle = \alpha_1\beta_1|\uparrow\uparrow\rangle + \alpha_1\beta_2|\uparrow\downarrow\rangle + \alpha_2\beta_1|\downarrow\uparrow\rangle + \alpha_2\beta_2|\downarrow\downarrow\rangle.$$

When we transform this it becomes

$$\begin{aligned}\hat{U}(\theta)|\Psi\rangle &= e^{i\theta/2}\alpha_1\beta_1|\uparrow\uparrow\rangle + e^{-i\theta/2}\alpha_1\beta_2|\uparrow\downarrow\rangle \\ &\quad + e^{-i\theta/2}\alpha_2\beta_1|\downarrow\uparrow\rangle + e^{i\theta/2}\alpha_2\beta_2|\downarrow\downarrow\rangle,\end{aligned}$$

which is no longer a product state for $\theta \neq m\pi/2$. The interaction has produced entanglement.

Quantum Gates and Circuits

We have seen that it is possible to build up new unitary operators by multiplying together some set of standard ones. This is analogous to the situation in classical logic, where *any* Boolean function can be built up from a set of standard functions of one or two bits, called *logic gates*: AND, OR, NOT, XOR, and so forth (see Chap. 2 appendix G and Chap. 5 appendix). We can similarly try to build up unitary transformations from a set of standard unitaries. We will call these *quantum gates*.

The simplest gate, affecting only a single qubit, is the NOT gate: $|0\rangle \leftrightarrow |1\rangle$. We see that this is also a familiar operator: the Pauli matrix $\hat{X}$. NOT is the only one-bit classical gate. But in quantum mechanics, there are far more possibilities. One important example with no classical analogue is the *Hadamard gate*:

$$\hat{U}_H = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}.$$

We write these unitaries in circuit diagrams with a convention similar to that of classical logic gates, as we will see in the next section.

We can also define two-qubit quantum gates. One example is the *controlled-not* (CNOT):

$$\begin{aligned} U_{\mathrm{CNOT}}|00\rangle &= |00\rangle, \\ U_{\mathrm{CNOT}}|01\rangle &= |01\rangle, \\ U_{\mathrm{CNOT}}|10\rangle &= |11\rangle, \\ U_{\mathrm{CNOT}}|11\rangle &= |10\rangle, \end{aligned}$$

There are, of course, an infinite number of possible two-qubit gates; but in practice, such unitaries are difficult to build. Fortunately, it turns out the CNOT, together with one-bit gates, can be used to build up *any* unitary.

When we combine standard unitary gates, we call the resulting unitary a *quantum circuit*. This is the most common way of representing a quantum computation, and is called the *circuit model* of a quantum computer. A *quantum algorithm* gives the construction of a quantum circuit to solve a particular problem.

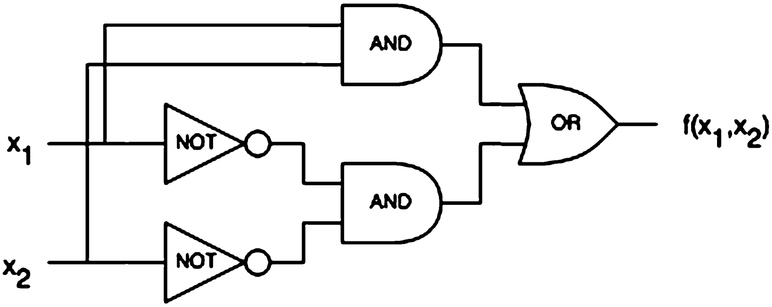
Quantum Algorithms

The Circuit Model

Boolean Circuits

It is very common to represent Boolean functions graphically as logic *circuits*. In this case, the logical values passed are indicated by *wires*, and the basic functions by *gates*. The figure gives an example of a Boolean circuit diagram.

In a computing context, the wires can be thought of as memory and each gate as an assignment. Note that the input variables were each used twice; we call this kind of duplication *fanout*. We also allow wires to cross each other, switching the relative positions of two variables. We call this a *crossover* or *swap*.



We can define several important quantifiers for a circuit. Define the *size* of a circuit to be the total number of assignments (or gates). The *depth* of a circuit is the *maximum number of gates along any path from the input to the output*. The circuit shown for $f(x_1,x_2)$ had size 5 and depth 3. Depth expresses how *parallelizable* a computation is. There is often a trade-off between depth and size.

Another sometimes-useful quantity is the *width*: the maximum number of wires leading to the output at any given time, not including the input variables. This gives a measure of the *space* used by a calculation. However, there is a strong trade-off between width and size; remarkably, *any* Boolean function can be calculated by a circuit with bounded width (though this has a large cost in size).

Complexity Theory and Circuits

Computational complexity theory (Chap. 12) is usually framed in terms of Turing Machines (TMs); but we would like to model computation in terms of Boolean circuits, which generalize more easily to the quantum case.

A TM can be given for a problem that will solve input instances of *any* size n. A given circuit, by contrast, will usually only solve instances of a single size n (or at best of size $\leq n$). We can make a connection to complexity theory by defining a *uniform family* of circuits for each problem. For every value of n, we define a circuit of size polynomial in n which solves all instances of the problem of size n. Moreover, there must be a TM which, given n as the input, outputs a description of the circuit in a time polynomial in n. It turns out that the problems for which there is a uniform family of circuits is equivalent to the complexity class P.

The requirement that the circuits can be generated by a TM is important. If we remove that assumption, we get a new class (called P/poly) of *nonuniform* circuit families, which includes P, but also problems not in P. Indeed, by some definitions this class includes noncomputable functions! Of course, for practical purposes, we would have no idea how to construct the successive members of a nonuniform circuit family.

There is no logical reason why naturally-occurring phenomena (e.g. quantum circuits) might not solve some noncomputable functions for us. Indeed, it has been speculated that some outcomes of the laws of physics may be noncomputable. No demonstration of this, however, is known.

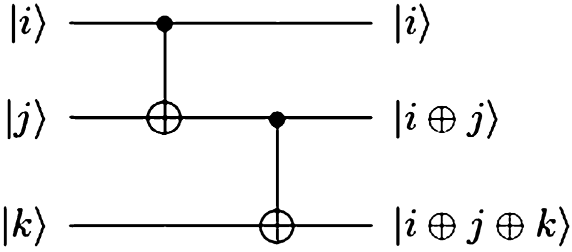
Quantum Circuits

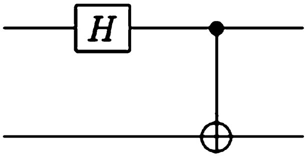
The notation for quantum circuits is quite analogous to the notation for classical Boolean circuits. A quantum circuit is a graphical representation of a unitary transformation for a quantum computer; the "wires" represent qubits, and the "gates" represent standard unitary transformations that act nontrivially on only one or two qubits at a time. Unlike an ordinary Boolean circuit, however, every quantum gate is *reversible*: the number of quantum bits in must equal the number of quantum bits out, and it is physically and logically possible to imagine running any gate backwards. (There can be exceptions if we allow destructive measurements as part of a circuit, but in principle we can always postpone these until the final readout of the computation.)

Also unlike the classical case, the qubits which enter a quantum circuit cannot be assumed to have individual states; in general, all of the bits may be in a joint, entangled state. Consider the circuit shown. This unitary takes computational basis states to computational basis states. If the input state is $|i\rangle|j\rangle|k\rangle$ the output state is $|i\rangle|i \oplus j\rangle|i \oplus j \oplus k\rangle$. If the input qubits are in a general state, we use linearity: write the state in terms of the computational basis and apply the circuit term by term. E.g.,

$$\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)|00\rangle \to \frac{1}{\sqrt{2}}(|000\rangle + |111\rangle).$$

Note that a product state has become entangled; this is the Greenberger-Horne-Zeilinger (GHZ) state.





This procedure is more complicated if we include gates which do not preserve the computational basis. For example, consider this circuit above involving a Hadamard gate and a CNOT. The Hadamard gate is applied only to the first qubit, while the CNOT affects both qubits. We can see how the basis states are transformed:

$$|00\rangle \rightarrow \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)|0\rangle \rightarrow \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle),$$

$$|01\rangle \rightarrow \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)|1\rangle \rightarrow \frac{1}{\sqrt{2}}(|01\rangle + |10\rangle),$$

$$|10\rangle \rightarrow \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)|0\rangle \rightarrow \frac{1}{\sqrt{2}}(|00\rangle - |11\rangle),$$

$$|11\rangle \rightarrow \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)|1\rangle \rightarrow \frac{1}{\sqrt{2}}(|01\rangle - |10\rangle).$$

This circuit takes computational basis states to entangled states—in fact, this set of entangled states has a name, the *Bell states*, and forms an orthonormal basis for the space of two qubits. However, if we apply this circuit to a general state, we must be careful to collect terms properly.

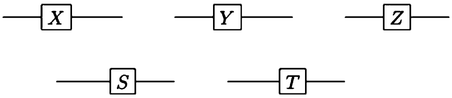
For instance,

$$\begin{aligned} \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)|0\rangle &= \frac{1}{\sqrt{2}}(|00\rangle + |10\rangle) \\ \rightarrow \frac{1}{2}(|00\rangle + |11\rangle &+ |00\rangle - |11\rangle) = |00\rangle. \end{aligned}$$

In this case, a product state is taken to a product state, in spite of the fact that this circuit can produce entanglement when applied to a basis state.

It is important to be able to translate a quantum circuit into the proper sequence of unitary transformations. The unitary appropriate to the gate is applied to the affected qubits, while all the other qubits are left unchanged.

We have so far seen the CNOT and the Hadamard gate. What are some other quantum gates? There is no standard list, but here are some common ones:



The Pauli gates X, Y, Z we know; the other two are

$$\hat{S} = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}, \hat{T} = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{pmatrix}.$$

Note that $\hat{T}$ is the square-root of $\hat{S}$, $\hat{S} = \hat{T}^2$, and $\hat{S}$ is the square-root of $\hat{Z}$, $\hat{Z} = \hat{S}^2 = \hat{T}^4$.

There are also common two-qubit and three-qubit gates. We have already seen the controlled-NOT gate; there is also the SWAP gate, which exchanges two bits (often indicated simply by crossing the wires in the diagram). Among three-qubit gates, a well-known example is the *Toffoli* gate, which is universal for classical reversible computation; quantum mechanically it exchanges $|110\rangle \leftrightarrow |111\rangle$. Another gate one sometimes encounters is the *Fredkin* gate, or controlled-SWAP: it exchanges $|110\rangle \leftrightarrow |101\rangle$.

We will also often see the controlled-*U* gate: an (n + 1)-qubit gate that applies the unitary operator Û on n bits if another control bit is in state $|1\rangle$, and not otherwise. This is typically a shorthand notation to represent a large sub-circuit. Other such sub-circuits are indicated by a block in which n qubits enter and exit, having undergone some unitary transformation $\hat{U}$.

Calculating Functions

A common component of many classical and quantum algorithms is calculating a function. In this case, we consider the n bits of input to represent an n-qubit integer x in binary notation $x_{n-1} \cdots x_1 x_0$. We will often use a short-hand notation in which $|x\rangle$ represents a state of n qubits: $|x\rangle = |x_{n-1} \cdots x_1 x_0\rangle$. Such an n-bit composite system is called a *quantum register*. The output is the value of the function $f(x)$, which we assume to be an m-bit integer in binary notation.

If $f(x)$ is an *invertible* function, then $m = n$ and the function must be a permutation of the numbers $0, \ldots, 2^n - 1$. In this case there is a unitary transformation $\hat{U}_f$ such that $\hat{U}_f|x\rangle = |f(x)\rangle$.

If f is not invertible (which is usually the case), there is no such $\hat{U}_f$, because such a transformation would not be unitary. (Remember, all unitary transformations are invertible.) What do we do then? Instead of using a single register to hold both the input and the output, we have two separate registers, and define a unitary transformation

$$\hat{U}_f(|x\rangle|y\rangle) \equiv |x\rangle|y \oplus f(x)\rangle,$$

in which $\oplus$ is bitwise-XOR. This transformation is clearly invertible. We now apply $\hat{U}_f$ to the input state$|x\rangle|0\rangle$ and get $|x\rangle|f(x)\rangle$ as the output. Most quantum algorithms contain a unitary transformation like this. Of course, to carry it out we actually need to construct a circuit to perform $\hat{U}_f$. If the function $f(x)$ has a classical circuit, we can make this reversible and translate it directly into a quantum circuit.

Scratch Bits and Ancillas

Just as in classical circuits, in some cases a circuit can be made more efficient by the use of extra bits, called "scratch bits," which are reset to zero at the end of the calculation. For instance, calculating a function $f(x)$ may be more conveniently done by first calculating some function $p(x)$, and then a function $g(p)$: $f(x) = g(p(x))$. We call $p(x)$ a *partial* result. We now have three registers: input, output, and scratch, and define two unitary transformations $\hat{U}_g$ and $\hat{U}_p$:

$$\hat{U}_p(|x\rangle|y\rangle|z\rangle) = |x\rangle|y\rangle|z \oplus p(x)\rangle,$$

$$\hat{U}_g(|x\rangle|y\rangle|z\rangle) = |x\rangle|y \oplus g(z)\rangle|z\rangle.$$

We see $\hat{U}_g\hat{U}_p(|x\rangle|0\rangle|0\rangle) = |x\rangle|f(x)\rangle|p(x)\rangle$. The scratch space can be re-used by inverting$\hat{U}_p$: $\hat{U}_p^{-1}\hat{U}_g\hat{U}_p(|x\rangle|0\rangle|0\rangle) = |x\rangle|f(x)\rangle|0\rangle$.

How do we do $\hat{U}_p^\dagger$? If a unitary is produced by a sequence of gates $\hat{U} = \hat{G}_N \cdots \hat{G}_1$, then obviously $\hat{U}^\dagger = \hat{G}_1^\dagger \cdots \hat{G}_N^\dagger$. Many common gates are their own inverses; for instance, the Hadamard, X, Y, Z, CNOT, SWAP, Toffoli and Fredkin gates. A circuit with only these gates can be inverted by it running backwards. If it involves other gates (such as the phase or π/8 gates), one must explicitly include their inverses. (E.g., $\hat{S}^\dagger = \hat{S}^3$.)

In the case of $\hat{U}_p^\dagger$, the case is even simpler. Since a second bitwise XOR undoes the first, in fact $\hat{U}_p$ is its own inverse. In this case, our sequence can be $\hat{U}_p\hat{U}_g\hat{U}_p$.

In addition to scratch bits, some extra ancilla bits may be necessary to make the overall circuit reversible. These ancillas are discarded at the end of the computation.

Oracles

Just as in classical computation, the idea of an *oracle* is useful in quantum computation. In the classical case, an oracle was a "black box" which input an n-bit number x and output a function $f(x)$. (We can make oracles reversible, just like any other classical circuit; for instance, it could input x and y and output x and $y \oplus x$.)

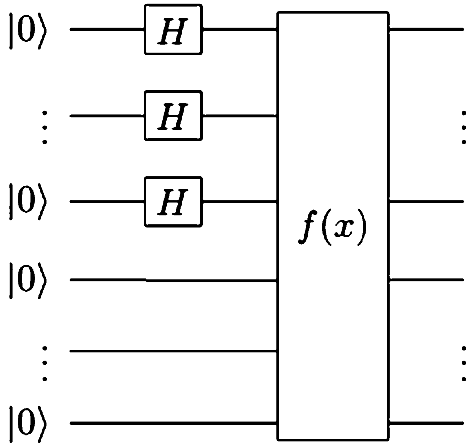
In the quantum case, an oracle is a black box which takes n qubits as input and performs a unitary transformation $\hat{U}$ on them. For instance, it could input two quantum registers, and perform the transformation $|x\rangle|y\rangle \rightarrow |x\rangle|y \oplus f(x)\rangle$.

However, there are other possibilities as well; for instance, it could do a transformation

$$|x\rangle \rightarrow (-1)^{f(x)}|x\rangle,$$

performing a phase shift conditioned on the value of the function $f(x)$.

Computations involving oracles usually center on determining some property of the function $f(x)$, or finding a value of x which has some particular value $f(x)$. In some cases, the fact that $f(x)$ is determined by a black box is important: if the function $f(x)$ were known, the problem would be solved. In other cases, however, this property is sufficiently nonobvious that knowledge of $f(x)$ makes no difference. In these cases, the difference between an oracle problem and an ordinary computation is somewhat blurred.



Just as in the classical case, we call each invocation of an oracle a *query*, and the number of queries needed to complete the circuit the *query complexity*.

Quantum Parallelism

So far, things don't look very different between classical and quantum circuits. What can we do with quantum bits that we can't with classical? Consider the circuit shown in the figure. The first n bits undergo Hadamard gates:

$$|0\rangle^n \rightarrow [(|0\rangle + |1\rangle)/\sqrt{2}]^n = \frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} |x\rangle.$$

This produces an equally-weighted superposition of all $|x\rangle$.

We then see what happens in the whole circuit. The input register is put in a superposition of all inputs, and together with the output register is passed to the oracle.

$$|0\rangle^n|0\rangle^m \rightarrow \left(\frac{1}{\sqrt{2^{\mathrm{n}}}}\sum_{\mathrm{x}=0}^{2^{\mathrm{n}}-1}|\mathrm{x}\rangle\right)|0\rangle^{\mathrm{m}} \rightarrow \frac{1}{\sqrt{2^{\mathrm{n}}}}\sum_{\mathrm{x}=0}^{2^{\mathrm{n}}-1}|\mathrm{x}\rangle|\mathrm{f(x)}\rangle$$

By a single call to the oracle, we have computed $f(x)$ for *every possible input* x! There is no classical analogue of this process, which Deutsch called *quantum parallelism*. It would seem that this capability should make quantum computers far more powerful than classical computers. But are they? By n Hadamard gates and a single oracle query, we have calculated $f(x)$ for every possible value of x; they are all contained in this massive superposition. But does it do us any good? What happens if we try to read the values out?

We do this by measuring the output register in the computational basis. If we do this, we get the result $f(x) = y$ with probability

$$p_y = \sum_{x,f(x)=y} 1/2^n.$$

We could do equally well by choosing a single x at random and calculating $f(x)$. This reflects a general principle: we cannot get more information out of measuring m qubits than m classical bits.

Let us see why this is so. The *Shannon entropy* bounds how much we can learn from a measurement:

$$S = -\sum_j p_j \log_2 p_j,$$

where j labels the outcomes. For a measurement of a D-dimensional system, there are at most D distinct outcomes, for which the maximum value of S is $\log_2 D$. In the case of m qubits, $D = 2^m$, so one can gain at most m bits of information. To learn $f(x)$ for all values of x would require $m2^n$ bits of information. Clearly this is out of the question.

So it might seem that, rather than being extraordinarily powerful, quantum parallelism buys you exactly nothing. But that, too, would be incorrect. One can acquire at most $m + n$ bits of information by measuring the input and output registers; but by being more subtle, one can do much better than just getting a random value of x and $f(x)$. One could instead, learn something about the function $f(x)$ *as a whole*. One could acquire up to $n + m$ bits of information giving some *property* of f, which might be difficult to acquire classically without making many queries to the oracle. Can we find an example of such a property?

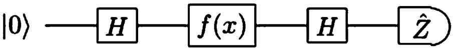
Deutsch's Problem

The simplest Boolean function $f(x)$ is a function of a single bit. There are exactly four such functions. Here they are:

Function	$f(0)$	$f(1)$
f_0	0	0
f_1	0	1
f_2	1	0
f_3	1	1

We see that f_0 and f_3 return the same thing regardless of the input x. We call these functions *constant*. By contrast, f_1 and f_2 return an equal number of 0's and 1's. We call these functions *balanced*. Suppose we have a classical oracle which inputs x and returns $f(x)$. How many queries are needed to determine if f is constant or balanced? (Answer: two queries.)

Consider the following circuit:



where the oracle does the transformation

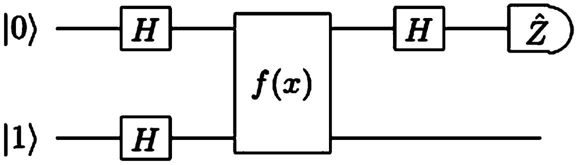
$$|x\rangle \to (-1)^{f(x)}|x\rangle.$$

We put the input qubit into state $(|0\rangle + |1\rangle)/\sqrt{2}$ and send it to the oracle. If f is *constant*, the state acquires an *overall* phase, and is returned to the state $|0\rangle$ by the second Hadamard. If f is *balanced*, the two terms acquire a relative phase of -1, and the qubit goes to $|1\rangle$. So a *single query* is sufficient to solve Deutsch's problem.

What if the oracle *isn't* the type that flips the phase of the state, but actually returns the value of $f(x)$:

$$|x\rangle|y\rangle \to |x\rangle|y \oplus f(x)\rangle.$$

We can still solve Deutsch's problem with only a single query. Consider this circuit:



Before the oracle query, the bits are in state $(|0\rangle + |1\rangle)(|0\rangle - |1\rangle)/2$. It is easy to check that applying the oracle gives

$$|\Psi\rangle = \Big((-1)^{f(0)}|0\rangle + (-1)^{f(1)}|1\rangle\Big)(|0\rangle - |1\rangle)/2.$$

So we again solve the problem by measuring the first bit *even though the value of f(x) was "put" in the second bit*. In QM the direction of information flow is basis dependent!

This was the first algorithm ever presented where a quantum computer outperformed a classical computer. This performance may not seem impressive—one query versus two—and the problem may be artificial. But the structure of the algorithm is worth noting:

1. The input and output registers start in computational basis states. ($|0\rangle$ and $|1\rangle$ in this case.)
2. By Hadamard gates, the input register is put in a superposition of all input values.
3. A unitary to evaluate $f(x)$ is done on both registers.
4. The input register is transformed again, and measured in the computational basis.

Quantum Fourier Transform

We now concentrate on a particular unitary transformation: the *quantum Fourier transform*. This is a discrete Fourier transform, not upon the data stored in the system state, but upon the state itself.

Let's look at the definition to make this a bit clearer. The discrete Fourier transform (DFT) of a discrete function $f_1, \ldots, f_N$ is given by

$$\tilde{f}_k \equiv \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} e^{2\pi ijk/N} f_j .$$

(See the book by Blum and Lototsky for a discussion of DFT.) The inverse transform is

$$f_j \equiv \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} e^{-2\pi ijk/N} \tilde{f}_k .$$

In the *quantum* Fourier transform, we do a DFT on the *amplitudes* of a quantum state:

$$\sum_j \alpha_j |j\rangle \rightarrow \sum_k \tilde{\alpha}_k |k\rangle ,$$

where

$$\tilde{\alpha}_k \equiv \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} e^{2\pi ijk/N} \alpha_j.$$

The question is: can we actually carry out this transform physically? This would be possible if there were a unitary operator $\hat{F}$ which transformed a state into its DFT:

$$|\tilde{\Psi}\rangle = \hat{F}|\Psi\rangle, \ \hat{F}^\dagger \hat{F} = \hat{I}.$$

First, we observe that the amplitudes $\tilde{\alpha}_k$ are linear in the original α_j. So there is a linear operator $\hat{F}$ which implements the transform. We can write it in outer product notation:

$$\hat{F} = \sum_{j,k=0}^{N-1} \frac{e^{2\pi ijk/N}}{\sqrt{N}} |k\rangle\langle j|.$$

It is easy to check that this does indeed produce the correct transformation $|\Psi\rangle \rightarrow |\tilde{\Psi}\rangle$, and that $\hat{F}$ is unitary: $\hat{F}\hat{F}^\dagger = \hat{I}$. The Fourier transform lets us define a new basis: $|\tilde{x}\rangle = \hat{F}|\mathrm{x}\rangle$, where $\{|x\rangle\}$ is the usual computational basis. This basis has a number of interesting properties.

Every vector $|\tilde{x}\rangle$ is an equally weighted superposition of all the computational basis states:

$$|\langle\tilde{x}|y\rangle|^2 = \langle y|\tilde{x}\rangle\langle\tilde{x}|y\rangle = \langle y|\hat{F}|x\rangle\langle x|\hat{F}^\dagger|y\rangle = \frac{e^{2i\pi xy/N}}{\sqrt{N}} \frac{e^{-2i\pi xy/N}}{\sqrt{N}} = \frac{1}{N}.$$

So if we think of the states $|x\rangle$ as being in a sense the most “classical,” then the states $|\tilde{x}\rangle$ are in a sense as “unclassical” as possible.

Recall that the Hadamard transform could also turn computational basis states into equally weighted superpositions of all states. But it left all amplitudes real, while the amplitudes of $|\tilde{x}\rangle$ are complex. And it was its own inverse, while $\hat{F} \neq \hat{F}^\dagger$. From the point of view of physics, the relationship of this basis to the computational basis is analogous to that between the *momentum* and *position* representations of a particle’s wavefunction.

Circuits for the Fourier Transform

At this point we will specialize to the case of n qubits, so the dimension is $N = 2^n$. We have seen that the quantum Fourier transform is a unitary operator. Therefore, by our earlier argument, there is a quantum circuit which implements it. However, there is no guarantee that this circuit will be efficient; a general unitary requires a circuit with a number of quantum gates exponential in the number of bits.

Very fortunately, in this case an efficient circuit does exist. (Fortunate, because the Fourier transform is at the heart of the most impressive quantum algorithms!) The key insight into designing a circuit for the Fourier transform is to notice that the states $|\tilde{j}\rangle$ can be written in a product form.

Let the binary expression for j be $j_1 j_2 ... j_n$, where $j = j_1 2^{n-1} + j_2 2^{n-2} + \cdots + j_n$. We also write binary fractions $0.j_1\, j_2 ... j_n = j_1/2 + j_2/4 + \cdots + j_n/2^n = j/2^n$. Then

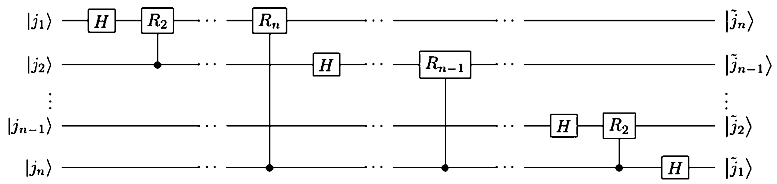
$$|\tilde{j}\rangle = 2^{-n/2}\big(|0\rangle + e^{2i\pi 0.j_n}|1\rangle\big)\big(|0\rangle + e^{2i\pi 0.j_{n-1}j_n}|1\rangle\big)\cdots\big(|0\rangle + e^{2i\pi 0.j_1\cdots j_{n-1}j_n}|1\rangle\big).$$

The unitary $|0\rangle \rightarrow (|0\rangle \pm \exp(i\theta)|1\rangle)$ is a Hadamard followed by a rotation of $\theta/2$ around the Z axis. In the expression above, the rotation depends on the values of the other bits. So we can build the Fourier transform out of Hadamards and controlled phase rotation gates.

Define the rotation

$$\hat{R}_k = \begin{pmatrix} 1 & 0 \\ 0 & e^{2i\pi/2^k} \end{pmatrix}.$$

A controlled-$\hat{R}_k$ gate does this if and only if a control qubit is $|1\rangle$ rather than $|0\rangle$. Putting these together with the Hadamards gives the following circuit:



This circuit performs the Fourier transform with the bits of the transformed state in *reverse* order. This circuit uses $n^2/2$ controlled-R gates, each of which can be produced with two CNOTs and two Z rotations. So the circuit as a whole uses n^2 CNOTS—definitely polynomial.

Periodic States

Suppose we are in N dimensions, and given a state of the form

$$|\phi\rangle = \sum_{n=0}^{N/r} c|\ell + nr\rangle,$$

where $|c| = \sqrt{r/N}$. We call this a *periodic* state with *period r* and *offset ℓ*.

What happens when we transform such a periodic state $|\phi\rangle \to |\tilde{\phi}\rangle$? The new state will have the form

$$|\tilde{\phi}\rangle = \sum_{m=0}^{r-1} \alpha_m |mN/r\rangle,$$

where $|\alpha_m| = \sqrt{1/r}$ for all m.

This state is also periodic, in the following sense: the α_m can have nontrivial phases, but they are all of equal weight; and the offset is *zero*.

Period Finding

We can exploit this fact to produce a quantum algorithm for *period finding*. Suppose $f(x)$ is a function from n-bit numbers to m-bit numbers. We have two quantum registers, an n-bit input register and an m-bit output register, and we prepare them in the state

$$|\Psi_0\rangle = \frac{1}{\sqrt{N}} \sum_{x=0}^{2^n-1} |x\rangle|0\rangle$$

using n Hadamard gates. (In general, there may be scratch bits as well, but we'll ignore that for now.)

We then apply a circuit that performs the unitary $\tilde{U}_f$:

$$\tilde{U}_f|\Psi_0\rangle = \frac{1}{\sqrt{N}} \sum_{x=0}^{2^n-1} |x\rangle|f(x)\rangle.$$

Suppose now that we measure the output register *only*, and get a particular value a. Then the input will be left in an evenly-weighted superposition of all x such that $f(x) = a$. If $f(x)$ is a periodic function with period r, then this state will look like this:

$$\frac{1}{\sqrt{N/r}} \sum_{n=0}^{N/r-1} |x_0 + nr\rangle|a\rangle,$$

where x_0 is the smallest value of x for which $f(x_0) = a$. The input register is in a periodic state.

Let us Fourier transform the input register. Then we'll get a state of the form

$$\sum_{m=0}^{r-1} \alpha_m |mN/r\rangle|a\rangle,$$

where$|\alpha_m| = \sqrt{1/r}$ and the particular phases of the α_m will depend upon the measured value of a.

What would happen if we now measured the *input* register? We will get one value mN/r for some m between 0 and $r - 1$. This by itself is not enough to tell us what N/r (and hence r) is. But let us run the algorithm d times. We will get a sequence of integers $m_1 N/r, \ldots, m_d N/r,$ which are all multiples of N/r. For a number of runs d which grows only moderately in N, we can be confident that with high probability, N/r is the *only* common factor of all the numbers.

Please note that we have implicitly assumed that $f(x) = f(y)$ *only* if $x = y + nr$—that is, except for the periodicity, this function has no repeated values. The reality can be more complicated, producing states which are superpositions of different periods with different weights. Fortunately, the functions we need for our algorithms have the simpler structure.

Greatest Common Divisor

Since the time of the ancient Greeks, an efficient algorithm has been known for finding the greatest common divisor (GCD) of two numbers: Euclid's algorithm. Suppose a and b are both multiples of some common divisor n, with $a > b$. Then if I divide a by b, the *remainder* a mod b will also be a multiple of n, and smaller than either a or b.

We repeat this procedure, this time with b and a mod b, and so on, until we reach the point where one of our numbers divides the other exactly. This number is the GCD of a and b. Since the numbers get smaller each time, we obviously must eventually find the answer; and more detailed analysis shows that it is computationally efficient.

If we take our numbers $m_1 N/r, \ldots, m_d N/r,$ and pairwise perform GCD on them, with high probability we will find the greatest common divisor of all of them to be N/r. This obviously gives us the value of r; and we have found the period of $f(x)$ by an efficient quantum algorithm (assuming that $\tilde{U}_f$ can be done efficiently).

In fact, going back over the algorithm, we find that the measurement of the output register is not really necessary: because different values of $f(x)$ are orthogonal, the associated periodic states of the input register cannot interfere with each other. Surprisingly enough, once we have calculated $f(x)$ by applying $\hat{U}_f$, we have no further use for the output register; if we like, we can throw it away!

Order-Finding

For integers x and N with no common factor, the *order* of x modulo N is the least positive integer r such that

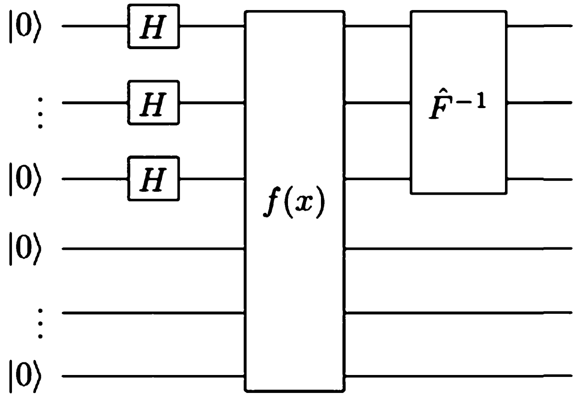
$$x^r = 1 \bmod N.$$

Obviously, $r \leq N$. (If not, the sequence of numbers $x^n =$ mod N for $n = 1,\ldots,r$ must all be distinct modulo N, which is impossible, since there are only N equivalence classes.)

Our problem is, given x and N, to find the order r of x modulo N. The description of the problem just requires the statement of x and N; we parametrize the size of the problem by $L = \log_2 N$, the number of bits needed to state N. No classical algorithm for order-finding is known which is polynomial in L.

Building a Quantum Algorithm

The first thing to notice is that we can define a function $f_x(n) = x^n \bmod N$. Since the order r means that $x^r = 1 \bmod N$, this means that $f_x(n+r) = x^{n+r} \bmod N = x^n x^r \bmod N = x^n \bmod N = f_x(n)$. So the function is *periodic* with period r. Moreover, the $f(n)$ must all be distinct for $0 \leq n < r$.



We can therefore build a circuit for order-finding based on our circuit for period-finding. We have divided the problem into two sub-circuits. The first performs the unitary $\hat{U}_{f_x}(|n\rangle|y\rangle) = |n\rangle|y \oplus f_x(n)\rangle$. The second performs the inverse Fourier transform on the input register. Both the input and the output registers start in the state $|0\rangle$, and the input register is put into a superposition of all $|n\rangle$ by Hadamard gates.

In fact, for this problem, it is better to have the second register start in the state $y = 1$, and create the unitary $\hat{U}'_{f_x}(|n\rangle|y\rangle) = |n\rangle|y \cdot f_x(n) \bmod N\rangle$. Note that this multiplication is invertible as long as x and N have no common factors; we would just multiply $yx^n \bmod N$ by x^{r-n} where r is the order of x. (Of course, we don't know what r is, but that doesn't change the fact that the multiplication is invertible in principle.) We call this unitary operator the circuit for *modular exponentiation.*

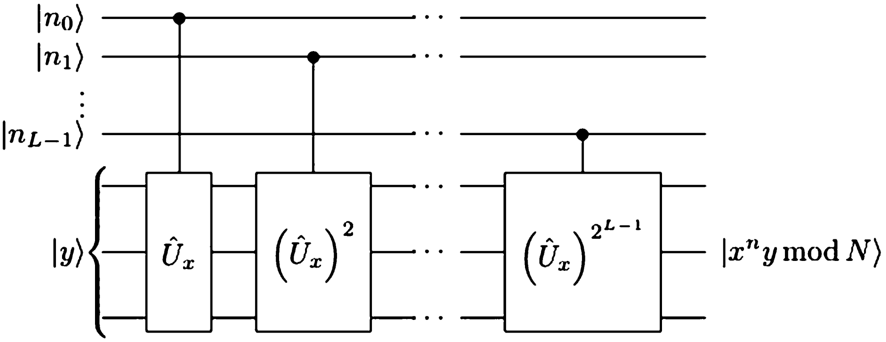
We can build modular exponentiation out of repeated applications of modular multiplication. Define a unitary operator $\hat{U}_x$ such that

$$\hat{U}_x|y\rangle = |xy \bmod N\rangle.$$

Let n be the argument of the modular exponential function, with L-bit binary representation $n_{L-1}n_{L-2}\cdots n_0 = n_{L-1}2^{L-1} + \cdots + n_0.\cdots$Then

$$x^n y \text{ mod } N = x^{n_{L-1}2^{L-1}} x^{n_{L-2}2^{L-2}} \cdots x^{n_0} y \text{ mod } N.$$

That is, we successively multiply y by x^{2^j} if $n_j = 1$, and by 1 otherwise. We can turn this into a quantum circuit involving a sequence of controlled unitary operations.



We build these subcircuits out of two other circuits: *modular multiplication* and *modular squaring*. These work as follows:

$$\hat{U}_m(|x\rangle) = |x\rangle|xy \,\text{mod}\, N\rangle, \quad \hat{U}_2|x\rangle = \left|x^2 \,\text{mod}\, N\right\rangle$$

Again, these are invertible as long as x has no common factors with N. (We don't care what they do in other cases, so it is possible to construct unitaries that do what we want.)

We also need a scratch register, which we start in the state $|1\rangle$. Let us write this first, so our full state is $|1\rangle|y\rangle$. The first thing we do is apply $\hat{U}_x$ to the scratch register:

$$|1\rangle|y\rangle \to |x\rangle|y\rangle.$$

We now apply $\hat{U}_2$ j times to perform $\hat{U}_x^{2^j}$:

$$|x\rangle y\rangle \to x^2 \,\text{mod}\, N\rangle|y\rangle \to |x^4 \,\text{mod}\, N\rangle|y\rangle \to \cdots \to |x^{2^j} \,\text{mod}\, N\rangle|y\rangle.$$

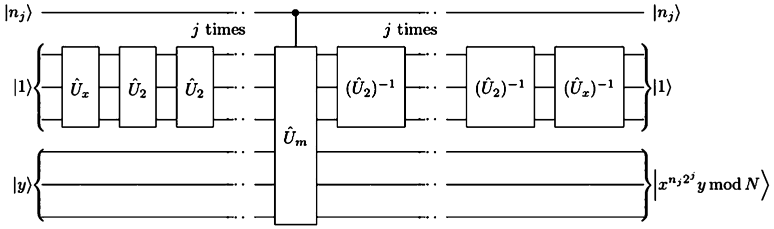
We then do modular multiplication $\hat{U}_m$:

$$|x^{2^j} \,\text{mod}\, N\rangle|y\rangle \to |x^{2^j} \,\text{mod}\, N\rangle|x^{2^j} y \,\text{mod}\, N\rangle.$$

Finally, we want to re-use the scratch space, so we "uncompute" x^{2^j} mod N by applying $\hat{U}_{2^j}^\dagger$ times and then $\hat{U}_x^\dagger$, making the whole transformation

$$|1\rangle|y\rangle \to |1\rangle|x^{2^j} y \,\text{mod}\, N\rangle.$$

The circuit for controlled-$\hat{U}_x^{2^j}$ looks like this:



(It would be more efficient to keep our partial results for $\hat{U}^{2^{j-1}}$for use with each successive n_j, but that does not alter the principle.) We can build modular squaring out of modular multiplication; and modular multiplication can be built out of modular addition and modular multiplication by two. Simple reversible circuits exist for both of these procedures.

We can now summarize the order-finding algorithm:

1. Prepare the input and output registers in states $|0\rangle$ and $|1\rangle$.
2. With Hadamard gates, put the input register in a superposition of all values $|n\rangle$.
3. Calculate the modular exponential function.
4. Do the inverse Fourier transform on the input register.
5. Measure the input register.
6. Find the order r using the GCD algorithm.

Our circuit uses $O(L^3)$gates. The input register requires $t = 2L + 1 + \log(1 + 1/2\varepsilon)$ bits to succeed with probability $p > 1-\varepsilon$.

Order-Finding and Factoring

While order-finding may seem of limited interest by itself, the problem of factoring large numbers reduces to order-finding for its most difficult cases. To state the problem concretely: given a (large) composite number N, we want to find one of its prime factors.

The algorithm proceeds in several steps, mostly eliminating special cases for which order-finding fails, but alternative efficient algorithms exist.

1. Check if N is even. If it is, obviously 2 is a factor.
2. Check if N is a power a^b for integers a and b. An efficient algorithm for this exists.
3. Choose a random integer x, $1 < x < N-1$. Calculate GCD(x,N) using Euclid's algorithm. If it is not 1, congratulations!
4. Use the order-finding algorithm to find the order r of x modulo N.

At this point, we use some number theory. If r is even, we calculate $x^{r/2} \bmod N$. If this is not $N-1 \equiv -1 \bmod N$ then we calculate $GCD\left(x^{r/2} \pm 1, N\right)$. If one of these gives a nontrivial factor, that is our answer. Otherwise the algorithm fails.

This may seem like a lot of conditions. But in fact, r has at least a 50% probability of being even and not having $x^{r/2} = -1 \bmod N$. If the algorithm fails, we just pick a new value of x and try again. The probability is overwhelming that we will succeed after only a few repetitions. This algorithm is $O(L^3)$ (from modular exponentiation and GCD). The best classical algorithm has a complexity $O(e^{L^{1/3}})$, which is superpolynomial.

Searching an Unordered Database

In searching for a needle in a haystack, one might hope that it pays to be systematic. Unfortunately, it does not.

Let $f(x)$ be a function whose argument is an integer $0 \leq x \leq N-1$, and which returns 1 for exactly one value x_1; for all other values of x, it returns zero. We can think of this function as being a database query, with the numbers x labeling record numbers or memory locations; we are searching for a particular record, and the function f tells us if we have found it.

From the point of view of computation, we consider f to be an oracle which can be repeatedly queried. How many queries are necessary before the value x_1 is found?

It is easy to see that an average of $N/2$ queries will be needed to find the "marked" record; and that there is no better algorithm than to just try one value of x after another until the desired location is found.

Because the problem has so little structure, every instance of the problem with the same value of N is equally difficult; and the order in which the queries are made is irrelevant. After a query, there is a probability of $1/N$ of finding the correct record; and if not, one is left with the same problem, with size $N-1$.

The Grover Algorithm

Can we do better with quantum mechanics? Let us assume that the oracle is a unitary transformation that takes $|x\rangle \rightarrow (-1)^{f(x)}|x\rangle$, i.e., it flips the sign of $|x\rangle$ if and only if $f(x) = 1$. Let us further assume, for the moment, that exactly one value $x_1$has $f(x_1) = 1$, and all others make $f(x) = 0$. (Later we will relax this assumption.)

We have seen that an oracle which takes $|x\rangle|y\rangle \rightarrow |x\rangle|y \oplus f(x)\rangle$ can be effectively "converted" to a phase oracle by preparing $|y\rangle$ in the state $(|0\rangle - |1\rangle)/\sqrt{2}$. So we will assume this phase form throughout.

Suppose that $N = 2^n$, so our system is n qubits, and we start in an evenly weighted superposition of all values x. (This can be prepared by n Hadamard gates.) The state is then

$$|\Psi\rangle = \frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} |x\rangle.$$

If we apply the oracle, the size of the amplitudes will remain unchanged, but the amplitude of the marked record will change sign.

The state at this point is

$$|\Psi'\rangle = \frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} (-1)^{f(x)} |x\rangle.$$

Applying the oracle again will just undo the previous application. Instead, we perform a unitary called "inversion about the mean."

$$|\Psi\rangle = \sum_x \alpha_x |x\rangle \rightarrow \sum_x (2\bar{\alpha} - \alpha) x\rangle, \quad \bar{\alpha} = \frac{1}{N} \sum_x \alpha_x.$$

It is easy to check that this is indeed unitary.

$$\hat{U} = \begin{pmatrix} 2/\mathrm{N} - 1 & 2/\mathrm{N} & \ldots & 2/\mathrm{N} \\ 2/\mathrm{N} & 2/\mathrm{N} - 1 & \cdots & 2/N \\ \vdots & & \ddots & \vdots \\ 2/\mathrm{N} & 2/\mathrm{N} & \cdots & 2/\mathrm{N} - 1 \end{pmatrix}, \quad \hat{U}^\dagger \hat{U} = \hat{I}.$$

If we now calculate the amplitudes of the new state, we see that the probability of the marked record has grown relative to the other peaks! By applying the oracle again, followed by "inversion about the mean," we can make the peak grow still further; but we cannot make the peak grow without limit. After a certain number of iterations, it will reach its maximum, and then start to shrink again.

Number of Iterations

We iterate the procedure until the marked record reaches its maximum, and then measure x; with high probability, we will find the correct value. How many iterations are needed to reach this maximum?

To answer this, we need to see that the algorithm corresponds to a rotation in a two-dimensional subspace. The initial state, $|\Psi\rangle$, is an even superposition of all basis states. The state we are aiming for is $|x_1\rangle$, which is one particular basis state. Both of these states lie in the subspace spanned by the two vectors $|x_1\rangle$ and

$$|\xi\rangle = \frac{1}{\sqrt{N-1}} \sum_{x, f(x)=0} |x\rangle.$$

These two vectors are orthogonal, and the initial state is $|\Psi\rangle = \sqrt{1/N}|x_1\rangle + \sqrt{(N-1)/N}|\xi\rangle$. The important thing to notice is that the iterations of the Grover algorithm do not move you out of this space. For a state $\alpha|x_1\rangle + \beta|\xi\rangle$, the oracle moves you to $-\alpha|x_1\rangle + \beta|\xi\rangle$, which is a reflection about the state $|\xi\rangle$.

We can re-write the "inversion about the mean" unitary as

$$\hat{U} = 2|\Psi\rangle\langle\Psi| - \hat{I},$$

where once again $|\Psi\rangle$ is our evenly-weighted initial state. Since $|\Psi\rangle$ lies in the subspace, it is obvious that $\hat{U}$ also leaves states in this subspace. In fact, $\hat{U}$ is also a reflection, this one about the state $|\Psi\rangle$. The product of two reflections is a rotation. Since the rotation must be independent of the state, clearly each rotation is by a constant amount. We want to rotate the state until it is as close as possible to $|x_1\rangle$.

Let us define the angle $0 < \theta < \pi/2$ such that

$$\cos(\theta/2) = \sqrt{(N-1)/N},$$

so that our initial state can be written

$$|\Psi\rangle = \sin(\theta/2)|x_1\rangle + \cos(\theta/2)|\xi\rangle.$$

After doing one iteration of the algorithm, the state is

$$\sin(3\,\theta/2)|x_1\rangle + \cos(3\,\theta/2)|\xi\rangle,$$

and after k iterations it is

$$\sin\left(\frac{2k+1}{2}\theta\right)|x_1\rangle + \cos\left(\frac{2k+1}{2}\theta\right)|\xi\rangle.$$

Each iteration rotates the state by θ; so we want to iterate until $(2k + 1)\theta \approx \pi$. Since θ is small, $\sin(\theta/2) = \sqrt{1/N} \approx \theta/2$. Finding the marked record requires a number of steps

$$k \approx (\pi/4)\sqrt{N}.$$

By contrast with classical search, which takes $O(N)$ queries, the quantum algorithm requires a number of oracle queries of $O\left(\sqrt{N}\right)$.

This is not an exponential gain in speed, so in one way it is less impressive than the factoring algorithm. On the other hand, this algorithm is *provably* better than the best classical algorithm; in the case of factoring, there is no proof that the best known classical algorithm is really optimal.

Building the Circuit

One part of the circuit is applying the oracle, which we already know how to do. What about this strange “inversion about the mean” unitary?

Notice that if we apply Hadamards to all n bits, the state transforms to

$$\frac{1}{\sqrt{2}^n}\sum_x \alpha_x|x\rangle \to \sum_{x,y}(-1)^{x\cdot y}\alpha_x|y\rangle,$$

where the Boolean dot product is

$$x\cdot y = (x_0\&y_0)\oplus(x_1\&y_1)\oplus\cdots\oplus(x_{n-1}\&y_{n-1}).$$

The important thing to note is that for $y = 0$, the new amplitude is

$$\alpha_0' = \frac{1}{\sqrt{2^n}}\sum_x \alpha_x = \overline{\alpha}.$$

This means that “inversion about the mean” is equivalent to the following procedure:

1. Apply Hadamards to all bits.
2. Flip the sign of the $|0\rangle$ state relative to all other basis states.
3. Apply Hadamards to all bits.

How do we carry out step 2? This is a controlled^{n-1}-Z gate, which can be built using $O(n)$ gates.

Applications

In principle, the Grover algorithm could be used to search a database. The database could be classical, but it would have to have a quantum interface.

A much more likely application would be to speed the solution of NP-complete problems. One method of solving decision problems (such as SAT) is to try each possible solution and check if it satisfies the decision criterion. Classically, this is like searching an unordered database, and requires a time of $O(2^n)$ for problems of size n.

By using Grover’s algorithm, the time could be reduced to $O\left(2^{n/2}\right)$, instead. This is, unfortunately, still exponential. But in practice, it could be enormously faster. More generally, we might use quantum searching to speed any program that checks many cases.

Decoherence and Error Correction

Decoherence

The description of quantum computers up to this point has been based on a major assumption: that the quantum computer behaves as an *ideal* quantum system. Unfortunately, real physical systems are imperfect. This leads to two effects that can derail a quantum computation.

The first, and easiest to understand, is imprecision in the control operations. In classical digital logic, the states are discrete, and the transformations between them (logic gates) are similarly discrete. By contrast, quantum states form a continuum. Suppose we wish to apply a quantum gate that rotates a state by θ. If the initial state is $|0\rangle$, we might get a state of the form $\cos(\theta)|0\rangle + \sin(\theta)|1\rangle$. But how can we be sure that we don't instead get the state $\cos(\theta')|0\rangle + \sin(\theta')|1\rangle$, where θ' is close, but not equal, to θ. Such small errors can accumulate as we apply multiple gates, to the point where the state we end up with is completely wrong.

The second effect is more subtle, but equally if not more important. This is *environmental decoherence*. In using the Schrödinger equation we are implicitly assuming that the system we described is isolated—that it does not interact with anything else in the world. In reality, however, this is never more than an approximation. Every quantum system interacts with its environment—nearby physical systems, which can include background electromagnetic fields, stray gas molecules and other impurities, vibrational modes of nearby solids, random incoming photons, and other systems that it is impossible to completely exclude.

Let us look at a toy model of decoherence, affecting a single qubit. Suppose this qubit is initially in some superposition state $|\Psi\rangle = \alpha|0\rangle + \beta|1\rangle$. However, it is not completely isolated; rather, it interacts with some environmental system. Suppose the initial state of the environment to be $|E\rangle$. The initial state of the complete system is then $|\Psi\rangle|E\rangle = (\alpha|0\rangle + \beta|1\rangle)|E\rangle$.

As the two systems interact, however, the state of the environment will change in a way that depends on the state of the system. After some time, the joint state will no longer be $|\Psi\rangle|E\rangle$, but will have become

$$a|0\rangle|E_0\rangle+\beta|1\rangle|E_1\rangle.$$

The system and the environment have become entangled, and the system no longer has a well-defined state of its own. This kind of decoherence is called *dephasing*.

Why is this important? Because this type of correlation destroys interference between the two components of $|\Psi\rangle$. Since quantum algorithms depend on interference, this kind of decoherence destroys their ability to function successfully.

Here is a second decoherence model, which looks more similar to a kind of classical error. Suppose again that the system and environment are in an initial state $|\Psi\rangle|E\rangle = (\alpha|0\rangle + \beta|1\rangle)|E\rangle$. But now we will suppose that the interaction between the state of the system and environment tends to alter the state of the system, so that after some time the state becomes $\sqrt{1-\varepsilon}(\alpha|0\rangle+\beta|1\rangle)|E_+\rangle+\sqrt{\varepsilon}(\alpha|1\rangle+\beta|0\rangle)|E_-\rangle$. In other words, the interaction between the system and environment tends to flip the value of the bit 0 ↔ 1. In a typical system, both of these decoherence effects can be present, and probably a variety of other (less familiar) ones as well.

The mathematical description of decoherence is beyond the scope of this chapter, but hopefully these examples give some intuition about how both decoherence and imprecision in control operations are a problem for quantum computers. Indeed, while these two sources of noise seem conceptually different, they have a unified mathematical description, and generally both are lumped together under the heading of *decoherence* or quantum noise.

To make matters worse, in a computer with many qubits the effects of decoherence on all the qubits accumulate. This derails a large computation more quickly than a small one. And the effect is compounded by the fact that large computations typically take longer than small ones as well, which gives decoherence longer to act. This problem could lead one to believe that beyond a certain rather small size, quantum computation would be impossible. Indeed, in the early 1990s when quantum computation first began to be seriously studied, many knowledgeable observers were confident that decoherence doomed any possibility of large-scale quantum computation.

Quantum Error Correction

Of course, classical computers are also subject to errors. At one time, it was thought that these errors might prevent large scale classical computation. But it was shown by John von Neumann and others that in fact this is not true: it is possible to do reliable computations using imperfect computers by making use of *error-correcting codes*. The simplest version of this idea is widely known. Instead of storing each bit once, one keeps redundant copies. So, for example, 0 and 1 are represented as 000 and 111. So long as errors are unlikely, they can be detected (if all three bits are not the same) and corrected (by majority vote). Much more sophisticated versions of this idea have led to the modern theory of error-correcting codes. An obvious thought, then is to try to protect quantum computers in the same way, with some form of *quantum error-correcting code*. In the early and mid-1990s, however, many people believed that quantum error correction was impossible.

This pessimism was based on two ideas. First, there is the point mentioned in the previous subsection, that quantum states form a continuum. This is different from classical bits, which are discrete. It is therefore possible for small, continuous errors to occur and accumulate. This is the effect that undermines large-scale analog computation.

Second, classical error-correcting codes are commonly understood to work by keeping *redundant copies* of the information that they protect. But by the famous *no-cloning theorem* of quantum mechanics, it is impossible to copy a quantum state. That is, there is no unitary transformation $\hat{U}$ that, given an arbitrary quantum state $|\Psi\rangle$ and another system in some standard state $|0\rangle$ will produce two copies of the arbitrary state, $|\Psi\rangle|\Psi\rangle$. The proof is quite elementary. So redundant storage of quantum information seems like a non-starter. Fortunately, both of these seemingly insurmountable problems turn out to be misconceptions, and quantum error-correcting codes are indeed possible. We will just sketch the main ideas here.

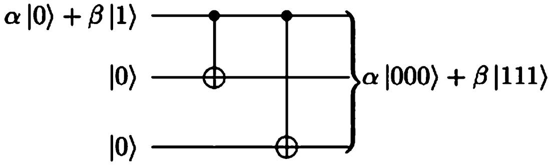
It is certainly true that quantum states form a continuum. But this is rather different than the continuous quantities used in analog computing. Consider two states $|0\rangle$ and $\sqrt{1-\varepsilon}|0\rangle + \sqrt{\varepsilon}|1\rangle$. One can continuously transform from one to the other. But an actual measurement of this qubit will not reliably distinguish them. If we measure the second state, with a high probability we will get the result 0—and then the system will be left in the state $|0\rangle$. With a small probability, we will get the result 1, and the state will be $|1\rangle$. This measurement has turned a continuous error into a discrete error. We can exploit this effect in error correction. A measurement will turn a continuous error in the state into a small probability of a discrete error. This property is called *discretization of errors*.

Of course, the problem is that a measurement will destroy the state just as much as an error would. Somehow we need to protect the information so that measurements can reveal the presence or absence of errors without disrupting the quantum state. The key to this comes from taking another look at classical error-correcting codes, but seeing them in a different way. Rather than keeping redundant copies of information, we can think of them as instead storing the information *nonlocally*, as a correlation over several bits. In this way, the codewords 000 and 111 can be thought of as storing a bit value as a correlation among the bits, rather than as multiple copies.

We can translate this directly into a quantum error-correcting code. Suppose we want to encode a single qubit $\alpha|0\rangle + \beta|1\rangle$; and suppose we know that the only kind of decoherence that occurs is bit flipping noise. We carry out the following encoding:

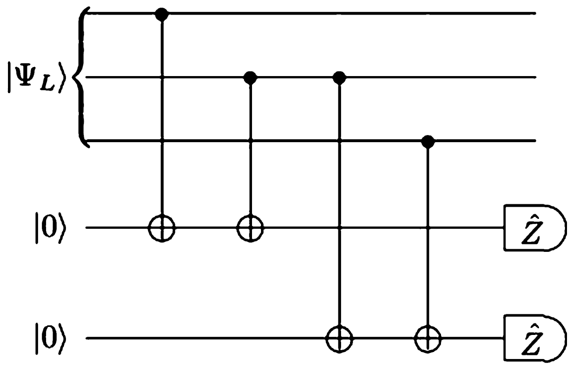
$$(\alpha|0\rangle + \beta|1\rangle)|00\rangle \rightarrow \alpha|000\rangle + \beta|111\rangle.$$

This encoding can be done by the simple quantum circuit shown.



$\alpha|0\rangle + \beta|1\rangle$
$|0\rangle$
$|0\rangle$
$\alpha|000\rangle + \beta|111\rangle$

Suppose a bit flip occurs on the first qubit. The state becomes $\alpha|100\rangle + \beta|011\rangle$. If we measured the three qubits, we could detect this error, but we would also destroy the superposition. So instead, we measure not the qubit values themselves, but the *parity* between qubits one and two, and between qubits two and three. The circuit below shows how this can be done.



These parity values reveal the presence and location of an error without destroying the superposition state. Moreover, once we know where the error is, we can undo it by applying a unitary $\hat{X}$ gate to the affected qubit.

Even better, we can handle continuous errors the same way. Suppose that the state $\alpha|000\rangle + \beta|111\rangle$ evolves continuously to a state $\sqrt{1-\varepsilon}(\alpha|000\rangle + \beta|111\rangle) + \sqrt{\varepsilon}(\alpha|100\rangle + \beta|011\rangle)$. After measuring the parities, we will detect either no error, with probability $1-\varepsilon$, or a single bit flip on qubit one with probability ε, which we can then correct. This is how discretization of errors can be exploited.

Of course, the assumption that only bit-flip errors can occur is quite unrealistic. This code would be useless against dephasing errors, or combinations of dephasing and bit flips. But remarkably, it turns out that quantum error-correcting codes exist that can protect against *general* errors, so long as they affect only a limited number of qubits at a time. Such codes were first discovered by Peter Shor, and independently by Andrew Steane, in 1996. The study of quantum error-correcting codes has rapidly advanced since then, and we now have a considerable understanding of the power and limitations of quantum error correction.

Fault-Tolerance and Threshold Theorems

This understanding has led to a series of theorems proving that *fault-tolerant quantum computation* is possible. While these theorems vary greatly in their details and assumptions, they all take a similar form: *If a quantum computation could be done with an ideal circuit of size N, then it can be done with probability approaching 1 by a quantum circuit of size N*·polylog*(N) that is subject to noise, provided that the error probability per gate is below a threshold* e_{th}. Because of this form, these theorems are known as *threshold theorems* (See, e.g., Aharonov and Ben-Or 1998; Gottesman 1998; Preskill 1998, 1999).

How big can e_{th} be? This is a difficult question to answer, and is highly dependent on the assumptions made about the quantum computer and the noise. Early estimates put lower bounds on e_{th} of 10^{-6} or 10^{-5}, which is dauntingly low.

Steady theoretical progress has improved this, to the point where some quantum computers have been projected to have thresholds of 10^{-3} or even 10^{-2}, a much more feasible-sounding error rate of 1%. In fact, we don't know how good thresholds could be in practice. The threshold theorems all make somewhat idealized assumptions; but at the same time, they often use antagonistic error models that are probably overly pessimistic. It is significant that no real nontrivial *upper* bounds on the threshold are known.

Typical assumptions in the threshold theorems are: independent errors between different qubits; the ability to do parallel operations; and the ability to perform gates between non-neighboring qubits. All of these assumptions can be relaxed to a certain degree, often at the cost of lowering the error threshold estimate. But analyzing a realistic quantum computer is sufficiently complicated, and the noise models are poorly enough known, that at present we really don't know what kind of threshold is achievable. Experimental progress over recent years has given some cause for optimism, however, as we discuss briefly in the next section.

Physical Implementations of Quantum Computers

Physical Requirements for Quantum Computation

The theoretical ideas that led to quantum computation were largely inspired by the rapid experimental progress of the 1970s and 1980s. Quantum optical systems, together with atom traps and ion traps, showed that individual quantum systems could be stored, addressed, and manipulated, while exactly obeying the predictions of quantum mechanics. But since the development of quantum information processing in the 1990s, it is fair to say that theory has vastly outstripped experiment in its progress. Many algorithms have been discovered; the entire field of quantum error correction has been invented; a vast, quantum extension to classical Shannon theory has been developed, drawing on quantum as well as classical resources. In the meanwhile, experimenters have struggled to control and measure more than a handful of quantum bits at a time.

This comparison tends to mask the tremendous progress that experimenters have made, especially in the last 10 years. Several different experimental systems now look promising enough that they may be scaled up, over the next few years, into quantum computers containing dozens or even hundreds of qubits, and capable of doing thousands of quantum gates.

To begin, then, we might ask: what properties make a physical system suitable to implement quantum computation? These requirements were presented in a very influential paper by David DiVincenzo, and are widely used to determine if a particular physical system could potentially be used to build a *scaleable* quantum computer (DiVincenzo 2000).

1. Existence of qubits (i.e., product structure). This is the essential requirement for scalability.
2. Controllable one- and two-qubit unitary gates. Given these, any unitary transformation can be built up, and all known quantum algorithms can be done efficiently. For fault tolerance, we would like to be able to perform quantum gates in parallel on different qubits.
3. Initializable in a known starting state. Without this, we cannot know what computation we are doing.
4. Ability to measure qubits in standard basis. This is obviously necessary to read out the results at the end; but it is also very useful in performing quantum error correction, as described in the previous section.
5. Very low intrinsic decoherence. Necessary to satisfy the threshold theorems for fault-tolerant quantum computation.

We will now briefly examine a couple of current experimental approaches to see how well they meet this list of requirements.

Ion Traps

One of the most powerful experimental developments of the last few decades was the development and improvement of two techniques: *laser cooling* and *electromagnetic traps*. By means of these, it is possible to cool small numbers of ions or atoms to nearly absolute zero and confine them at a precise location in a vacuum chamber, where they can be repeatedly probed by properly-tuned lasers. This is the closest we have come to being able to achieve the type of quantum measurements envisioned by von Neumann in the 1930s: projective measurements that probe the state of the system without destroying it. Using these techniques, it may be possible to achieve all of the DiVincenzo criteria. Initially proposed in Cirac and Zoller 1995, a scalable architecture was proposed in Kielpinski et al. 2002.

Qubits

An atom (or ion) consists of a positively charged nucleus with some number of negatively charged electrons which are bound to the nucleus by the Coulomb force. In an ion, these charges do not cancel, so the ion has a net charge; either some electrons have been stripped away or added.

Since the electrons are attracted to the nucleus, they “try” to be as close to it as possible; however, they repel each other. Also, no two electrons can be in exactly the same state due to the Pauli Exclusion Principle. The arrangement of electrons which minimizes the energy subject to these constraints is the *ground state* $|g\rangle$. If the atom or ion acquires extra energy, an electron may be “kicked” into a higher orbit, making the atom *excited*. The lowest-lying excited state will be labeled $|e\rangle$.

Quantum Gates

One way of exciting an ion is by *resonant driving*. The principle is the same as pumping up a swing: the electrons have natural resonant frequencies. Because the electrons are charged, a periodic electric field will exert a periodic force on the electron; if the period matches the resonance frequency, the atom will be excited; it will make a transition from the initial state to a higher energy state.

This kind of periodic electric field can be provided by a laser tuned to the appropriate frequency. The proper laser frequency is determined as follows: the energy of a single photon is $\hbar\omega$, where ω is the light frequency. This energy must equal the *difference* between the two atomic energy levels.

The atom absorbs a single photon and becomes excited. (If the laser is left on, the atom will actually make a transition back to the starting state, re-emitting the absorbed photon. This is called *stimulated emission*.)

If the laser is tuned away from a resonance frequency the rate of transition rapidly diminishes. Because the level spacing of most atoms and ions is not very even, this means we can drive particular transitions with great specificity. For instance, it is possible to drive a transition that will happen if the atom is in state $|e\rangle$ but not in $|g\rangle$. Also, certain transitions may be *forbidden* by other conservation laws (such as parity and angular momentum).

We can now see how to build a qubit out of a trapped ion. We identify the two states $|0\rangle \equiv |g\rangle$ and $|1\rangle \equiv |e\rangle$ as our basis vectors, and carry out one bit gates by driving transitions with appropriately-tuned lasers.

Note that because $|g\rangle$ and $|e\rangle$ do not have the same energy, there will be a constantly-accumulating relative phase between them:

$$\alpha|0\rangle + \beta|1\rangle \to \alpha|0\rangle + \mathrm{e}^{i\Delta Et/\hbar}\beta|1\rangle.$$

We must keep track of this phase as we perform our quantum gates. In our description we will just automatically undo this phase with extra Z rotations. A description like this is called a *rotating frame*.

To produce two-bit quantum gates, we make use of an additional degree of freedom to couple the internal states of the ions: the *motion* of the ions. To understand how this works, we need to make a brief digression to talk about the simple harmonic oscillator. The Hamiltonian of a harmonic oscillator is

$$\hat{H} = \frac{\hat{p}^2}{2m} + \frac{m\omega^2\hat{x}^2}{2}, \text{ where } [\hat{p},\hat{x}] = -i\hbar.$$

This system is solved by finding the energy eigenstates $|n\rangle$. These have evenly-spaced eigenvalues $\hat{H}|n\rangle = \hbar\omega(n + 1/2)|n\rangle \equiv E_n|n\rangle$.

A single ion in a trap acts like a harmonic oscillator to a good approximation. What about multiple ions? The motion of the ions can be decomposed into *normal* *modes*. For N ions, there are N normal modes; each of these acts like a separate harmonic oscillator with its own characteristic frequency.

By laser cooling, the ions in the trap are reduced to their *motional ground-state*: each of the normal modes is in the state $|0\rangle$. It is possible to excite transitions of one of the normal modes to an excited state by driving the ions at the resonance frequency ω.

However, there is a much more interesting possibility. It is possible to excite a normal mode *conditional on the electronic state of one of the ions*. We can use this as a building block to construct two-bit gates, using the vibrational normal mode as a kind of "communication bus." We thus see that we can do both one-qubit and two-qubit quantum gates.

Measurement

This would all be pointless if we were unable to measure the state of our qubits. We can do this by resonant driving as well. The key is to drive a transition from one of the basis states to an *unstable* excited state. This state will rapidly decay, emitting a photon in a random direction. This transition can be driven repeatedly, scattering many photons in a short time. These scattered photons can be detected by an ordinary CCD camera. If the ion is in state $|1\rangle$ it will glow visibly when illuminated by a properly-tuned laser. If in state $|0\rangle$ it will remain dark. This is a near-perfect projective measurement.

The availability of high-quality projective measurements is one of the most attractive features of the ion-trap quantum computer. For many implementations, measurement is a challenging technological problem.

Decoherence

These are the main intrinsic sources of decoherence for the linear ion trap:

1. *Spontaneous emission*. Since the $|e\rangle \to |g\rangle$ transition is forbidden, the $|e\rangle$ state has a long lifetime; however, by more complicated processes it can still decay. More significant is the possibility of decay from an excited state in the performance of a gate; by detuning and using metastable states, this can be kept under control.
2. *Leakage*. To act like a qubit, the ion must remain in the subspace spanned by $|g\rangle$ and $|e\rangle$. There is always a possibility of an accidental transition to states outside this space. This is mainly controlled by tuning the lasers very precisely, and choosing ions whose transition frequencies are not too close together.
3. *Heating*. Until recently, this was the dominant source of decoherence. Because the ions are charged, they are very sensitive to the presence of stray electric and magnetic fields. These can lead the normal modes "heating up," which interferes with two-bit gates. Recently, a great deal of progress has been made, by carefully designing the equipment and actively cooling the ions using *sympathetic cooling* of extra ions in the trap.

4. In addition there are the usual problems of precision. To work as described, the lasers must be very precisely tuned, and the intensity and duration of pulses tightly controlled. There are also difficulties if the lasers are not tightly enough focused on individual ions (though careful design can get around this).

Recent Developments

The scheme of packing all the ions into a single trap is inherently limited. Because only a single normal mode (or at most a few) can be used at a time, it is impossible to do many two-bit gates in parallel. Cooling must be turned off while gates are performed, which makes error rates grow. It is difficult to focus a laser down onto a single ion without accidentally affecting its neighbors. With a single trap, scalable quantum computing is impossible.

To avoid this problem, recent experiments have used a new architecture. Instead of one trap, there are multiple trapping regions, each holding a few ions. When a two-bit gate is performed, the ions holding the two qubits are physically moved into the same trapping region; their normal mode is cooled into the ground state, and the gate is performed. They can then be moved back into storage.

These traps are also being dramatically shrunk down in size. Instead of the three-dimensional arrangement of electrodes used in earlier experiments, all of the electrodes in these new experiments are laid out on a flat surface, using photolithographic techniques. This allows many trapping regions to be established close to each other, so that ions can be transferred between them without their electronic states begin disturbed.

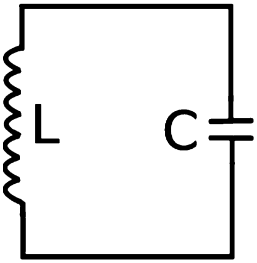
Another important breakthrough was the use of "sympathetic cooling." In addition to the ions used to store qubits, additional ions of a different atomic species are included. These ions do not store qubits, but because they have different resonance frequencies, they can be laser-cooled continuously without interfering with the other ions.

Performance

Experiments at present can handle 4–8 ions in a trap quite well—for certain special purposes, as many as 12 ions have been manipulated, though this is definitely not general-purpose. They can do hundreds of one- and two-bit gates before losing coherence. Progress is also being made in designs which permit parallel operations on many qubits at once, which would make concatenated codes and fault-tolerant design possible. The gates at present are done with a fidelity of 95–99%. Ion traps are widely considered the mostly likely to succeed in building a medium-scale quantum computer in the not-too-distant future (if all goes well, within the next 5–10 years). At present, ion traps are the leading contenders for scalable quantum computing.

Superconducting Qubits

LC Circuits



A type of circuit that is well-known from classical circuit theory is the *LC circuit*, in which an inductor and a capacitor cause oscillations in the flux of a circuit loop. The energy function for this circuit can be written

$$H = \frac{Q^2}{2C} + \frac{\Phi^2}{2L}, \ \omega = \frac{1}{\sqrt{LC}}.$$

Here, Q is the charge on the capacitor, Φ is the magnetic flux in the inductor, and C and L are the capacitance and inductance.

We can make circuits smaller and smaller, so that Q and Φ become noncommuting quantum observables, like position and momentum:

$$[\hat{\Phi}, \hat{Q}] = \hat{\Phi}\hat{Q} - \hat{Q}\hat{\Phi} = i\hbar.$$

But any quantum effects are masked by resistance, which causes decoherence.

We can get rid of this decoherence by going to very low temperatures where our circuit is *superconducting*. The electrons become bound into *Cooper pairs*, and all resistance vanishes. The phase is well-defined and the system is a quantum harmonic oscillator.

Josephson Junctions and Superconducting Qubits

A superconducting LC circuit would not work as a qubit, because (like any harmonic oscillator) it has infinitely many evenly-spaced energy levels; resonant driving does not let us single out two levels to use as $|0\rangle$ and $|1\rangle$, nor perform gates on this subspace alone.

We can fix this problem by adding a *Josephson junction* to the circuit. This is a thin insulating barrier between two ends of superconductor. Classically this would act as either an open circuit, or at best a capacitor; but in the quantum regime, Cooper pairs can tunnel across the gap, so a current can flow. The effect of including a Josephson junction is to add a nonlinear term to the harmonic oscillator potential.

The potential with the added Josephson junction takes this form:

$$U(\Phi) = E_J\left[1 - \cos\left(2\,\pi\frac{\Phi_{\rm ex} - \Phi}{\Phi_0}\right)\right] + \frac{\Phi^2}{2L'},$$

where $\Phi_0 = h/2e$ is the *quantum of flux*, $\Phi_{\rm ex}$ is the external bias flux, and E_J is the Josephson energy, which is proportional to the critical current through the junction.

The behavior of the circuit is determined by the applied bias flux $\Phi_{\rm ex}$ and the ratio E_J/E_C where E_C is the capacitor charging energy $E_C = e^2/2C$. This leads to three different designs for superconducting qubits. The three qubit types are the *charge* qubit, the *flux* qubit, and the *phase* qubit.

- *Charge qubit* (Cooper-pair box): omits the inductance L and uses the two lowest levels of the cosine potential for $|0\rangle$ and $|1\rangle$. These represent the absence and presence of a single extra Cooper pair on the "island" of superconducting metal.
- *Flux* or *persistent-current qubit*: uses a double-well potential representing current circling clockwise or counterclockwise.
- *Phase qubit*: uses bias to produce unequal wells. The lowest two levels of the *higher* well are $|0\rangle$ and $|1\rangle$; the lower well is used to make measurements.

Quantum Gates

The level separations in superconducting qubits are typically in the range 5–10 GHz (microwaves). Single-bit quantum gates can be done using *resonant driving*, just as for the ion trap, but at much lower frequencies.

A one-qubit gate is performed by applying a pulse of RF, tuned to the precise resonance frequency between (e.g.) $|0\rangle$ and $|1\rangle$. This can produce an X or Y gate. A Z gate can be done by using the energy splitting between $|0\rangle$ and $|1\rangle$ to produce a relative phase. The pulse can be produced by an external signal generator, and carried to the qubit by wires. RF engineering can be done with great precision; these pulses have very exact timing and frequency control.

As for two-qubit quantum gates, superconducting qubits are (relatively) macroscopic objects—a typical qubit is μms or 10s of μms in size. Therefore it is straightforward to couple nearby qubits either capacitively or inductively. A disadvantage of this scheme, however, is that only neighboring qubits can be coupled, and the coupling is always turned on. (This is not necessarily a fatal problem, however.)

A more flexible scheme uses a *tuneable coupler*. This is an inductance that can be controlled by externally applied fields. It can also couple qubits that are not directly adjacent. This has been demonstrated experimentally for phase qubits.

It is also possible to couple two qubits indirectly, by coupling each of them to the same *transmission line resonator*. This acts like a bus, similar to the vibrational mode in an ion trap quantum computer. By resonant driving, one can conditionally excite a microwave photon in the resonator from one qubit, absorb and re-emit it from another, and reabsorb it at the first, to produce an effective gate for two qubits.

One can also couple two qubits by tuning them close to resonance, but far enough that no actual photon is emitted or absorbed. This is slower than the first scheme, but less sensitive to photon loss. Transmission line resonators can couple qubits that are physically far apart—it has been demonstrated at distances of millimeters.

Measurement

Measurement of superconducting qubits is one of the most difficult problems, because it is difficult to do without allowing in outside noise, causing decoherence.

- Charge qubits can be measured by charge detectors.
- Flux qubits have been measured in different ways, but generally by inductive coupling to a nearby microscopic device, such as a SQuID.
- Phase qubits can be measured destructively by lowering the energy barrier to induce tunneling into the deeper well. This produces a detectable current, but destroys the state. Nondemolition measurement has been demonstrated more recently by coupling to a transmission line resonator to produce a state-dependent phase shift.

Decoherence

Because superconducting qubits are macroscopic objects, they are subject to many sources of noise. Early qubits had very short decoherence times—on the order of 1 ns. Improved design and fabrication have greatly improved this, bringing lifetimes up to ~1 μs.

The exact sources of decoherence are not completely known. Experimentalists believe that the qubits couple to microscopic degrees of freedom that arise from defects in the bulk substrate, in the insulating layer, and in imperfections in the circuit construction.

Measurement can also open up new sources of noise—for instance, thermal noise from an external amplifier can leak down the transmission line.

State of the Art

Superconducting qubits have made the most impressive progress in the last few years, raising their decoherence times by orders of magnitude and demonstrating entanglement between two and three qubits. It is also possible to couple qubits that are physically far apart. Experimenters are currently planning to scale up their systems to include 5–10 qubits and multiple transmission line couplers (See, e.g., Martinis et al. 2002; Niskanen et al. 2007; Lupascu et al. 2007 and many others).

On a different scale, and pursuing a different methodology, a private company, D-Wave, has built a chip with 128 qubits. This is not a general-purpose quantum computer, however, but rather an *adiabatic* quantum computer.

Other Systems

Many other systems are being actively pursued for quantum computation. Here are a few of the most promising.

- *Liquid-state NMR*. The most impressive experiments so far have been done using liquid-state NMR at room temperature, including a demonstration of Shor's algorithm using six quantum bits. However, this system does not really satisfy the DiVincenzo criteria, and thus is unlikely to lead to true scalable quantum computing.
- *Solid-state NMR*. There are various schemes, using the spins of nuclei in crystals; or using the nuclei of phosphorus atoms embedded in silicon, coupled via electron spins.
- *Quantum dots*. These are tiny devices that serve as "wells" to hold electrons. The spin of an electron can serve as the qubit, or the location of the charge in one of two dots. Like superconducting qubits, this work draws on the large amount of technological expertise at building ever-smaller solid-state devices.
- *Linear optical QC*. It is possible for photons to serve as qubits (as we saw in looking at quantum cryptography), and they have very low rates of decoherence, but it is hard to make them interact; this makes it difficult to build two-bit gates. It is possible to build *probabilistic* two-bit gates, however, by passing photons through interferometers and measuring some of the outputs.

It is still too early to know which technology may eventually win out (For a recent review of different experimental implementations and their current state of the art, see Ladd et al. 2010).

Further Topics

This chapter has only scratched the surface of quantum computing. Many other algorithms have been discovered since the pioneering work of Deutsch, Shor, and Grover. Moreover, the practical issues of building a quantum computer have been the subject of intensive research. A new theory of quantum error correction has been developed, proving that it is possible (in principle) to construct quantum computers that can scale up to problems of any size, provided that the intrinsic noise levels per operation are sufficiently low. Experimental efforts have concentrated on developing hardware capable of achieving those low noise levels, and combining these elements into larger systems. This work is in many ways still in its infancy, but the improvement is already very impressive.

Other research has tried to place quantum computers into the hierarchy of computational complexity classes. Whole new classes have been developed for quantum computers, and computer scientists have looked for problems that separate the performance of quantum and classical algorithms. This work, too, is only at the beginning.

Finally, the use of quantum systems for communications has also become a vibrant field of its own. Shannon's information theory has been supplemented by the use of new, essentially quantum resources: quantum channels, shared entangled states, and others. This has led to a new, rich topic of quantum information theory, including puzzling protocols such as quantum teleportation, quantum key distribution, and quantum superdense coding. The field of quantum information processing continues to grow and flourish in a remarkable way; it is a very exciting time.

Reference

D. Aharonov and M. Ben-Or, in Proc. 29th Ann. ACM Symp. on Theory of Computing, 176 (ACM, New York, 1998).

P. Benioff, Journal of Statistical Physics 29, 515–546 (1982).

C.H. Bennett and G. Brassard, in Proc. IEEE International Conference on Computers Systems and Signal Processing, Bangalore India, December 1984, 175–179 (IEEE, 1984).

E.K. Blum and S.V. Lototsky, "Mathematics of physics and engineering," section 6.4 (World Scientific, Singapore, 2006).

J. I. Cirac and P. Zoller, Phys. Rev. Lett. 74, 4091–4094 (1995).

D. Deutsch, Proc. Roy. Soc. London, Ser. A 400, 97 (1985).

D. Deutsch, Proc. Roy. Soc. London, Ser. A 425, 73 (1989).

D.P. DiVincenzo, Fortschr. Phys. 48, 771–783 (2000).

R. P. Feynman, Int. J. Theor. Phys. 21, 467 (1982).

D. Gottesman, Phys. Rev. A 57, 127 (1998).

L.K. Grover, in Proc. 28th Annual ACM Symposium on the Theory of Computing (STOC 96), 212 (ACM, New York, 1996).

D. Kielpinski, C. Monroe, and D.J. Wineland, Nature 417, 709–711 (2002).

T.D. Ladd, F. Jelezko, R. Laflamme, Y. Nakamura, C. Monroe and J.L. O'Brien, Nature 464, 45 (2010).

A. Lupascu, S. Saito, T. Picot, P. C. de Groot, C. J. P. M. Harmans, and J. E. Mooij, Nat. Phys. 3, 119–125 (2007).

Y. Manin, Sovetskoye Radio (Moscow, 1980).

J.M. Martinis, S. Nam, J. Aumentado, and C. Urbina, Phys. Rev. Lett. 89, 117901 (2002).

Nielsen and Chuang, "Quantum Information and Quantum Computation" (Cambridge University Press, Cambridge, 2000).

A. O. Niskanen, K. Harrabi, F. Yoshihara, Y. Nakamura, S. Lloyd, and J. S. Tsai, Science 316, 723–726 (2007).

J. Preskill, Proc. Roy. Soc. Lond. A454, 385–410 (1998).

J. Preskill, Physics Today, (June 1999).

P.W. Shor, in Proc. 35th Annual Symposium on the Theory of Computer Science, edited by S. Goldwasser, 124 (IEEE Computer Society Press, Los Alamitos, CA, 1994).

Chapter 15
Numerical Thinking in Algorithm Design and Analysis

Shang-Hua Teng

Numerical Thinking

To me, numerical analysis is one of the most fascinating fields in computing. It is at the intersection of computer science and mathematics; it concerns subjects that can be either continuous or discrete; it involves algorithm design as well as software implementation; and it has success throughout engineering, business, medicine, social sciences, natural sciences, and digital animation fields. Its community has both theorists and practitioners, who often respect and admire each other's work – in fact, in this area there are many practitioners who are also great theoreticians. Its objectives to solve larger and larger problems have pushed the envelope of computer science, particularly in the advancement of computer architectures, compiler technologies, programming languages, and software tools. Its collaborative culture and genuine need to share data and information among scientists and engineers have led a physicist to make a connection between the hypertext idea and the Internet protocols to create the world wide web. Many pioneers in computing including John von Neumann, Alan Turing, Claude Shannon, Richard Hamming, James Wilkinson, Velvel Kahan, and Gene Golub contributed to this field, not to mention the foot prints left by many great minds before them, e.g., Newton's method, Lagrange interpolation, Gaussian elimination, Euler's method, Jacobi iteration, and Chebyshev polynomials.

Inspite of its glamorous history, many computer science students may have completed their undergraduate degrees or even Ph.D. degrees without taking any class in Numerical Analysis after taking the linear algebra class during their sophomore year. I hope this will change because during the last two decades, we have seen several striking successes in the use of numerical methods & concepts

S.-H. Teng (✉)
Department of Computer Science, Viterbi School of Engineering,
University of Southern California, Los Angeles CA, USA
e-mail: shanghua@usc.edu

E.K. Blum and A.V. Aho (eds.), *Computer Science: The Hardware, Software and Heart of It*, DOI 10.1007/978-1-4614-1168-0_15,

ranging from web search (PageRank) & data mining (Latent Semantic Indexing), to digital animation (Photoreal Digital Actor) & image processing (Wavelets).

"Numerical thinking" is the process of discovering useful connections between numerical analysis and other fields in computing. In this article, I would like to share with you two examples in which numerical thinking has significantly impacted the research of my collaborators and myself in algorithm design and analysis. The first example is *smoothed analysis*, an algorithm analysis framework towards explaining the good practical behaviors of algorithms and heuristics. The second example is the *Laplacian paradigm*, an emerging algorithmic framework for designing nearly-linear time network analysis algorithms that process massive graphs. I hope these two examples will encourage more researchers to use numerical thinking in their work. I also hope that more and more of our students and scholars will be exposed to the principle of numerical analysis.

Smoothed Analysis of Algorithms

Applying stability analysis to algorithms, smoothed analysis (Spielman and Teng 2004) provides a framework aiming to rigorously explain the practical success of several algorithms and heuristics that have poor worst-case complexity. The rapid progress in this area has underscored the promise and importance of perturbation in algorithm analysis. Perturbation has also been instrumental in several recent breakthroughs in algorithm design and complexity theory. It is used in the solution of problems ranging from mesh generation to mathematical optimization to algorithmic game theory.

In section "Algorithm Design and Analysis with Perturbations", I survey some of these advances. I will focus on the role of perturbations in both algorithm design and algorithm analysis.

The Laplacian Paradigm

In section "The Laplacian Paradigm: Emerging Algorithms for Massive Graphs", I discuss an emerging paradigm for designing efficient graph algorithms. This paradigm, which we will refer to as the Laplacian Paradigm, is built on a recent suite of nearly-linear time primitives in spectral graph theory developed by Spielman and Teng (2008a,b,c), especially their solver for linear systems $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is the Laplacian matrix of a weighted, undirected n-vertex graph and $\mathbf{b}$ is an n-place vector.

In the Laplacian Paradigm for solving a problem (on a massive graph), we reduce an optimization or computational problem to one or multiple linear algebraic problems that can be solved efficiently by applying the nearly-linear time Laplacian solver. So far, the Laplacian paradigm has already had some successes.

It has been applied to obtain nearly-linear-time algorithms for applications in semi-supervised learning, image processing, web-spam detection, eigenvalue approximation, and for solving elliptic finite element systems. It has also been used to design faster algorithms for generalized lossy flow computation and for random sampling of spanning trees. We hope that the Laplacian Paradigm will become a useful tool in the development of faster algorithms for solving fundamental problems in combinatorial optimization (e.g., the computation of matchings, flows and cuts), in scientific computing (e.g., spectral approximation), in machine learning and data analysis (such as for web-spam detection and social network analysis), and in other applications that involve massive graphs.

Acknowledgements

This article combines and refines my two earlier presentations:

- "Algorithm Design and Analysis with Perturbations," at the *Fourth International Congress of Chinese Mathematicians* (2007).
- "The Laplacian Paradigm: Emerging Algorithms for Massive Graphs," at the *7th Annual Conference of Theory and Applications of Models of Computation* (2010).

The first presentation, using smoothed analysis as the main example, focused more on applying numerical principles to algorithm design & analysis. The latter, centered at the Laplacian Paradigm, is more about the mutual impact of graph algorithms and numerical algorithms. I hope the materials from these two presentations together will help to highlight the usefulness of numerical thinking in algorithm design and analysis.

Both smoothed analysis and the Laplacian paradigm have been a joint work with Dan Spielman of Yale University.

This research is in part supported by NSF grants CCF 0964481 (AF: Medium: Smoothed Analysis in Multi-Objective Optimization, Machine Learning, and Algorithmic Game Theory) and CCF 1111270 (AF: Large: Algebraic Graph Algorithms: The Laplacian and Beyond).

I would like to thank Ed Blum for inviting me to write a chapter for his book *Computer Science: The Hardware, Software and Heart of It*. I thank Kanak Agrawal, Fei Sha, and Ed for their valuable editorial help.

Algorithm Design and Analysis with Perturbations

Perburbation has been part of algorithm design and analysis from the very beginning. In numerical computing, perturbation is central to stability analysis of numerical algorithms. In geometric applications, perturbation is one of the main tools for handling degeneracy and for designing high-quality finite-precision algorithms. Data perturbation has also been successfully used in machine learning, data security, optimization, and designing approximation algorithms.

In the examples to be discussed in this section, perturbation is the key to smoothed analysis, where it is used to model the imprecision that is often inherent in practical inputs. Smoothed analysis has been applied to understand the performance of several algorithms and heuristics for applications ranging from mathematical programming (Spielman and Teng 2004), clustering (Arthur and Vassilvitskii 2006), scheduling (Schäfer et al. 2003), machine learning (Blum and Dunagan 2002; Kalai et al. 2009), linear system solving (Sankar et al. 2005), motion planing (Damerow et al. 2003), and local search (Englert et al. 2007). The rapid progress in smoothed analysis has demonstrated the promise and importance of perturbations in algorithm analysis.

In this section, I would like to illustrate the usefulness of perturbations in both algorithm analysis and algorithm design. In addition to smoothed analysis, I will discuss the role of perturbations in several recent algorithmic breakthroughs. Some of these algorithmic results were inspired by the success of smoothed analysis, while others were obtained independently. Their approaches are similar. They often use the following property of the underlying problems:

> An input instance can be transformed so that the transformed instance admits some perturbations whose solutions can be found efficiently. Moreover, the solutions of the perturbed instances are useful for solving the original instance.

I will discuss three examples. The first example, from Cheng et al. (2000), uses perturbations to solve a long-term open question in three-dimensional mesh generation. It uses the following geometric fact: If the Delaunay triangulation of a set of three-dimensional periodic points has a bounded radius-edge ratio, then there is a perturbation of the weights of these points so that their weighted Delaunay triangulation has a bounded aspect-ratio (See section “Well-Shaped Mesh Generation” for the definitions of radius-edge ratio, aspect ratio, and weighted Delaunay triangulation). The second example, from Kelner and Spielman (2006), gives a perturbation-based simplex algorithm. This algorithm is the first provable polynomial-time simplex-like algorithm for linear programming. It uses the following geometric property: If the righthand of the inequalities of a bounded polytope is perturbed, then the projection of the perturbed polytope onto a randomly chosen two-dimensional subspace has polynomial shadow size, with high probability. The third example, from Sankar et al. (2005), provides a robust Gaussian elimination algorithm for solving linear systems. It uses the observation that the smallest singular value of a perturbed matrix is not likely to be too close to 0.

Finally, I will review a recent result of Chen et al. (2009a) on the approximation and smoothed complexity of the two-player Nash equilibrium. In this example, the attempt to prove that the smoothed complexity of the classic Lamke–Howson algorithm for two-player Nash equilibria is polynomial, has eventually led to the discovery of a fundamental result about the approximation complexity of Nash equilibria. This result can also be extended to characterize the smoothed and approximation complexity of market equilibria.

Smoothed Analysis

In “Challenges for Theory of Computing: Report for an NSF-Sponsored Workshop on Research in Theoretical Computer Science” (1999), its authors Condon, Edelsbrunner, Emerson, Fortnow, Haber, Karp, Leivant, Lipton, Lynch, Parberry, Papadimitriou, Rabin, Rosenberg, Royer, Savage, Selman, Smith, Tardos, and Vitter wrote:

> While theoretical work on models of computation and methods for analyzing algorithms has had enormous payoff, we are not done. In many situations, simple algorithms do well. Take for example the Simplex algorithm for linear programming, or the success of simulated annealing of contain supposedly intractable problems. We don’t understand why! It is apparent that worst-case analysis does not provide useful insights on the performance of algorithms and heuristics and our models of computation need to be further developed and refined. Theoreticians are investing increasingly in careful experimental work leading to identification of important new questions in algorithms area. Developing means for predicting the performance of algorithms and heuristics on real data and on real computers is a grand challenge in algorithms.

Smoothed analysis is largely motivated by this grand challenge concerning the fundamental discrepancy between the traditional theoretical analyses and the practical performance of algorithms.

The simplex method has been effectively used since the 1950s to solve optimization problems in numerous industrial applications. It solves a mathematical program given by

$$\begin{array}{ll}\textbf{maximize} & \mathbf{c}^{\mathrm{T}}\mathbf{x} \\ \textbf{subjectto} & \mathbf{A}\mathbf{x} \leq \mathbf{b},\end{array} \tag{15.1}$$

where $\mathbf{A}$ is an m by n matrix, $\mathbf{b}$ is an m-place vector, and $\mathbf{c}$ is an n-place vector.

The solution space $\{\mathbf{Ax} \leq \mathbf{b}\}$, if feasible, defines a convex polytope. A linear programming problem may have three types of answers. If $\{\mathbf{Ax} \leq \mathbf{b}\}$ is empty, then the program does not have a solution; if $\{\mathbf{Ax} \leq \mathbf{b}\}$ is not empty but unbounded in the direction of $\mathbf{c}$, then the solution is unbounded, otherwise, it has a finite solution at an extreme vertex of the polytope $\{\mathbf{Ax} \leq \mathbf{b}\}$.

The simplex method usually has two phases. In Phase I, it determines the type of the answer. If the program is feasible with a bounded solution in the direction of $\mathbf{c}$, Phase I also produces an initial extreme vertex $\mathbf{v}_0$ on the polytope $\{\mathbf{Ax} \leq \mathbf{b}\}$. In Phase II, it iterates starting from $\mathbf{v}_0$, where in the i-th iteration ($i \geq 1$), if $\mathbf{v}_{i-1}$ has a neighboring vertex $\mathbf{v}_i$ whose objective value $\mathbf{c}^T\mathbf{v}_i$ is better than $\mathbf{c}^T\mathbf{v}_{i-1}$, then enter iteration $i+1$, or else, terminate with $\mathbf{v}_{i-1}$ as the final result.

Despite its excellent practical behavior, it is well known that there are linear programming instances that force the simplex method to take exponential time to complete its iterative search.

How to Model Real Data and How to Measure Practical Performance?

Although intuitively simple, data modeling for practical computing is in fact extremely challenging, and often nearly impossible. Part of the difficulty comes from the fact that most algorithms are designed to handle a large range of data instances rather than just a particular instance. Yet, most of the time, users of an algorithm care more about the performance of the algorithm on instances they encounter, while the subsets or the distributions of instances encountered usually vary from user to user.

For example, when a financial analyst needs to apply an optimization algorithm A to design a portfolio for financial data x, she is mostly interested in the performance of A on data x. However, as the financial data changes from customer to customer, she may need to gain more understanding of the performance of A on the various instances that she receives. Furthermore, if there are ten optimization algorithms $A_1, \ldots A_{10}$ for portfolio design, how should she decide which one to buy?

Let Ω denote the set of all possible financial data instances. Let $\phi_A(x)$ denote the *instance-based measure* of the performance of an algorithm A on $x \in \Omega$. The complete measure of the performance of A is given by its *performance landscape*

$$[\ \phi_A(x) \quad |x \in \Omega\ ].$$

Note that if Ω is finite, the performance landscape can be viewed as a $|\Omega|$ dimensional vector.

Now, suppose the financial analyst receives the performance landscapes

$$[\ \phi_{A_1}(x) \quad |x \in \Omega\ ] \quad \ldots \quad [\ \phi_{A_{10}}(x) \quad |x \in \Omega\ ],$$

of all ten optimization algorithms, how should she decide which one is better?

- Should she use the performances of these algorithms on the instance of her most valued customer?
- Should she use the best performances of these algorithms over a set of benchmark instances or all possible instances?
- Should she use the worst performances of these algorithms over a set of benchmark instances or all possible instances?
- Should she use the average performances of these algorithms over a set of the benchmark instances or all possible instances?
- Should she assume that her data arises from certain distribution?
- Should she assume that her data possesses certain property?
- How should she choose her benchmark data? Should she not include any data that rarely occurs in practice?

Using the worst performance of an algorithm over the domain of all possible data as measurement, the traditional *worst-case analysis* can be conducted even when we have little information about the real data. Such analysis gives an absolute performance guarantee regardless of the instances that may emerge in practice, since the worst-case guarantee is completely independent of any particular instance. For the sake of illustration, imagine a tourist comes to LA and seeks advice on the amount of elevation he should be prepared to go up to while climbing the Hollywood hill, we answer, "no more than 29,035 ft" (because we know that the height of the peak of Mount Everest, the tallest peak on the earth is 29,035 ft). If he then asks, "How about Mt Hood? I plan to go there too." We can still answer "no more than 29,035 ft!"

The worst-case analysis is particularly useful for those algorithm analyses where one can show that the algorithm under consideration has a desirable worst-case behavior. However, when the worst-case performance of an algorithm is far from being desirable and the worst-case instance rarely occurs in practice, then we could be overly pessimistic about the algorithm.

Thus, to better measure the performance of an algorithm on real data, we need better understanding and a more refined model of input data. *Average-case analysis* was an important step in theoretical computer science to model the input. In an average-case analysis, one assumes there is a distribution π over the input Ω, and measures the performance of an algorithm A by its expected performance $\mathrm{E}_{x \in \pi(\Omega)}[\phi_A(x)]$. The major challenge in conducting meaningful average-case analyses, however, is to choose a meaningful distribution π that simultaneously lends to rigorous mathematical analysis and is close to the practical distribution of inputs (Spielman and Teng 2009).

To overcome the difficulty of average-case and worst-case analyses, theoreticians sometimes conduct *property-based performance analysis* (Lipton et al. 1979; Miller et al. 1997, 1998; Spielman and Teng 1996; Mitzenmacher and Vadhan 2008; Balcan et al. 2009) by assuming that inputs have certain property. For example, in social network analysis, one may assume that the input graph satisfies some powerlaw degree distribution, while in the finite-element analysis, one may assume that the input graph is a well-shaped mesh. By assuming input instances satisfy certain property, better performance guarantees are usually possible. For instance, in our earlier example, if we know that the tourist is only considering mountain climbing in the US, we could give him a better answer, "no more than 20,320 ft (the height of Mount McKinley in Alaska)."

Smoothed analysis takes a step forward in combining property-based analysis with worst-case & average-case analyses in its attempt to model real data. It focuses on the imprecision property of real data.

To illustrate the basic imprecision property, let us consider the following example: If one asks, "What is IBM's current stock value?", one may check its current market value and return with $168.28. But an hour later, it may become $164.75 and another hour later, $170.62.

So, how should we model IBM's stock value? Well, this value varies from measurement to measurement. But if someone were to say that IBM's stock value is $50 or $800, then most likely you would say that it can't be true. On the other hand, if someone said that IBM's stock value is $140 or $200, you may think it more believable.

One way to express this simultaneous uncertainty and certainty in our view of a stock price is to write it as the sum of two numbers:

$$s_{\text{IBM}} = \bar{s}_{\text{IBM}} + r_{\text{IBM}}$$

where $\bar{s}_{\text{IBM}}$ is determined by the intrinsic business value of IBM, and r_{IBM} is a random number that models the imprecisions introduced by the trading market. In this case, $\bar{s}_{\text{IBM}}$ could be equal to \$165, while r_{IBM} is a random variable from certain distribution.

Suppose the input data for portfolio design consists of the stock values of n companies, in smoothed analysis, we assume the value of each stock can be expressed as the sum of two numbers, where the first number is given by the company itself, while the second number models the random imprecision from a stochastic distribution when the market makes its assessment.

In other words, in the smoothed model, an input is neither completely random nor completely arbitrary – inputs are generated from a two-step process: In this first step, an instance is generated and in the second step, the instance from the first step is slightly perturbed. The perturbed instance is the "real data", and is the input to the algorithm. In the section below, we define a measure of performance under the smoothed model.

I would like to emphasize that smoothed analysis is only a step in our attempt to model real data in explaining the behavior of algorithms in practice. Modeling real data and measuring practical performance continue to remain grand challenges in computing.

Smoothed Complexity

Let set D_n denote the set of all input instances whose input size is n. Suppose $Q(\mathbf{x})$ is a measure on input $\mathbf{x}$. For algorithmic studies, $Q(\mathbf{x})$ may be the time complexity, the space complexity, the parallel complexity, or the cache complexity of an algorithm when $\mathbf{x}$ is its input. For non-algorithmic studies such as in matrix theory, $\mathbf{x}$ could be a matrix and $Q(\mathbf{x})$ could be its condition number. In our example from the last section, $\mathbf{x}$ could be a mountain and $Q(\mathbf{x})$ could be the height of its peak.

The traditional *worst-case measure* is then given by:

$$W[Q_n] = \max_{\mathbf{x} \in D_n} Q(\mathbf{x}).$$

To define the traditional *average-case measure*, one first determines a distribution of the inputs and then computes the expected measure assuming inputs are drawn from this distribution. Supposing $\mathcal{S}$ is a distribution over D_n, the average-case measure according to $\mathcal{S}$ is

$$AVG_{\mathcal{S}}[Q_n] = E_{\mathbf{x} \in_{\mathcal{S}} D_n}[Q(\mathbf{x})],$$

where we use $\mathbf{x} \in_{\mathcal{S}} D_n$ to denote that $\mathbf{x}$ is randomly chosen from D_n according to distribution $\mathcal{S}$.

To define smoothed complexity, we first need to determine a perturbation model that best captures the randomness and imprecision in the formation of inputs. We then assume that inputs are subject to random perturbations according to this perturbation model.

For problems arising in numerical computing, optimization, and computational geometry, we can often assume $D_n = \mathbb{R}^n$. The popular perturbation models include Gaussian and uniform perturbations. Let $\bar{\mathbf{x}} \in \mathbb{R}^n$. A σ-Gaussian perturbation of $\bar{\mathbf{x}}$ is a random vector $\mathbf{x} = \bar{\mathbf{x}} + \mathbf{g}$, where $\mathbf{g}$ is a Gaussian random vector of variance σ^2. A σ-uniform perturbation of $\bar{\mathbf{x}}$ is a random vector $\mathbf{x}$ chosen uniformly from the ball of radius σ centered at $\bar{\mathbf{x}}$. For combinatorial problems, a commonly used model is the Boolean perturbation. Let $\bar{\mathbf{x}} = (\bar{x}_1, \ldots, \bar{x}_n) \in \{0,1\}^n$ or $-1, 1^n$ A σ-Boolean perturbation of $\bar{\mathbf{x}}$ is a random string $\mathbf{x} = (x_1, \ldots, x_n) \in \{0,1\}^n$ or $-1, 1^n$, where $x_i = \bar{x}_i$ with probability $1 - \sigma$.

Definition 1 (Smoothed Measure). *Let $\mathcal{R} = \cup_{n,\sigma} R_{n,\sigma}$ be a family of perturbations over $D = \cup_n D_n$, where $R_{n,\sigma}$ defines for each $\bar{\mathbf{x}} \in D_n$ a perturbation distribution of $\bar{\mathbf{x}}$ with magnitude σ. Suppose $Q(\mathbf{x})$ is a measure of input $\mathbf{x}$. The smoothed Q_n measure is then*

$$\max_{\bar{\mathbf{x}} \in D_n} \left(E_{\mathbf{x} \leftarrow R_{n,\sigma}(\bar{\mathbf{x}})} [Q(\mathbf{x})] \right), \tag{15.2}$$

where $\mathbf{x} \leftarrow R_{n,\sigma}(\bar{\mathbf{x}})$ means $\mathbf{x}$ is chosen according to distribution $R_{n,\sigma}(\bar{\mathbf{x}})$.

When time complexity is the main concern, the central question in smoothed analysis naturally is whether an algorithm has polynomial smoothed complexity.

Definition 2 (Polynomial Smoothed Complexity). *Given a problem P with input domain $D = \cup_n D_n$. Let $\mathcal{R} = \cup_{n,\sigma} R_{n,\sigma}$ be a family of perturbations where $R_{n,\sigma}$ with magnitude σ. Let A be an algorithm for solving P and $T_A(\mathbf{x})$ be the time complexity for solving an instance $\mathbf{x} \in D$. Then algorithm A has polynomial smoothed complexity if there exist constants n_0, σ_0, c, k_1 and k_2 such that for all $n \geq n_0$ and $0 \leq \sigma \leq \sigma_0$,*

$$\max_{\bar{\mathbf{x}} \in D_n} \left(E_{\mathbf{x} \leftarrow R_{n,\sigma}(\bar{\mathbf{x}})} [T_A(\mathbf{x})] \right) \leq c \cdot \sigma^{-k_2} \cdot n^{k_1}. \tag{15.3}$$

The problem P is in smoothed polynomial time with perturbation model $\mathcal{R}$ if it has an algorithm with polynomial smoothed complexity.

The smoothed complexity of an algorithm is measured in terms of input size as well as magnitude of the perturbations. As the perturbation magnitude increases continuously starting from 0, the smoothed complexity interpolates between the worst-case and average-case complexity (Spielman and Teng 2004).

Smoothed Analysis of the Simplex Algorithm

Spielman and Teng (2004) introduced smoothed analysis to resolve the discrepancy between the poor worst-case complexity and the impressive practical performance of simplex algorithms. They analyzed the stability of the complexity landscape of simplex algorithm with shadow-vertex pivoting rule, assuming input instances are subject to slight random perturbations.

In (Spielman and Teng 2004), the following smoothed model is used, For any $\bar{\mathbf{A}}, \bar{\mathbf{b}}, \bar{\mathbf{c}}$, the perturbations of the linear program defined by $(\bar{\mathbf{A}}, \bar{\mathbf{b}}, \bar{\mathbf{c}})$ is

$$\max \quad \mathbf{c}^T\mathbf{x} \qquad \text{subject to} \quad \mathbf{A}\mathbf{x} \leq \mathbf{b},$$

where $\mathbf{A}$, $\mathbf{b}$, and $\mathbf{c}$, respectively, are obtained from $\bar{\mathbf{A}}, \bar{\mathbf{b}}, \bar{\mathbf{c}}$ by a Gaussian perturbation of variance $(||\bar{\mathbf{A}}, \bar{\mathbf{b}}, \bar{\mathbf{c}}||_F \cdot \sigma)^2$, where $|\,|\,(\mathbf{A}, \mathbf{b}, \mathbf{c})\,|\,|_F$ is the square root of the sum of squares of the entries in $\mathbf{A}, \mathbf{b}$, and $\mathbf{c}$.

A Key Perturbation Lemma

Let $\mathbf{A}$ be a σ-Gaussian perturbation of an $m \times n$ matrix $\bar{\mathbf{A}}$ with $||\bar{\mathbf{A}}||_F \leq 1$ and $\mathbf{1}$ be the m-vector all of whose entries are equal to 1. Then the polytope $\{\mathbf{x} : \mathbf{A}\mathbf{x} \leq \mathbf{1}\}$ is always feasible with $\mathbf{0}$ as a feasible point. For any two n-vectors $\mathbf{c}$ and $\mathbf{t}$, the projection of the polytope $\{\mathbf{x} : \mathbf{A}\mathbf{x} \leq \mathbf{1}\}$ on the two-dimensional plane spanned by $\mathbf{c}$ and $\mathbf{t}$ is a polygon and is called the *shadow* of the polytope onto the plane spanned by $\mathbf{c}$ and $\mathbf{t}$. We denote this shadow by $\mathbf{Shadow}_{\mathbf{t}\,\mathbf{c}}\mathbf{A}$. This shadow is a random polygon.

The analysis of Spielman and Teng hinges on the following perturbation property that the expected size of the projection of a perturbed polytope onto a two-dimensional subspace is polynomial in number of constraints, dimension, and the inverse of the magnitude of the perturbation.

Lemma 1 (Smoothed Shadow Size). *For any $m \times n$ matrix $\bar{\mathbf{A}}$ with $||\bar{\mathbf{A}}||_F \leq 1$, let $\mathbf{A}$ be a σ-Gaussian perturbation of $\bar{\mathbf{A}}$. For any two n-place vectors $\mathbf{c}$ and $\mathbf{t}$*

$$E_{\mathbf{A}}\left[\left|\mathbf{Shadow}_{\mathbf{t},\mathbf{c}}(\mathbf{A})\right|\right] = O((mn^3)(\min(\sigma^2, 1/(n\log m))^{-3}).$$

Smoothed Complexity of the Shadow-Vertex Method

By proving that the smoothed complexity of the simplex algorithm is polynomial in the size of the linear program and the inverse of the magnitude of perturbations, Spielman and Teng showed that the imprecision of the practical inputs alone would guarantee the good performance of the simplex algorithm.

Theorem 1 (Smoothed Complexity of a Simplex Algorithm). *For any* $\bar{\mathbf{A}} \in \mathbb{R}^{m\times n}, \bar{\mathbf{b}} \in \mathbb{R}^m, \bar{\mathbf{c}} \in \mathbb{R}^n$ *with* $||\bar{\mathbf{A}}, \bar{\mathbf{b}}, \bar{\mathbf{c}}||_2 \le 1$, *let* **A**,**b**,**c** *be Gaussian perturbations of* $\bar{\mathbf{A}}, \bar{\mathbf{b}}, \bar{\mathbf{c}}$, *respectively, of variance* σ^2. *Then there exists a two-phase simplex algorithm to solve the linear program defined by* **A**,**b**,**c**, *in expected time polynomial in m,n and 1/σ.*

While the formal analysis of this theorem and Lemma 1 requires some mathematical subtlety, the intuition behind the analysis is in fact simple. In Lemma 1, when feasible, the projection $\mathbf{Shadow_{t,c}}(\mathbf{A})$ of the polytope $\mathbf{x} : \mathbf{Ax} \le \mathbf{1}$ on the two-dimensional plane spanned by **c** and **t** is a polygon. To prove that $\mathrm{E}_{\mathbf{A}}(\mathbf{Shadow_{t,c}}(\mathbf{A}))$ is polynomial in m, n and $1/\sigma$, we show that due to the noise in **A**, the probability that this random polygon has a short edge is small. Moreover, we prove that it is unlikely that two adjacent edges in this random polygon meet at an angle too close to π.

We then apply Lemma 1 to bound the complexity of each phase in the simplex algorithm, where each phase defines a shadow polygon. In the proof, we need to carefully handle the probability dependency between these two phases. In (Vershynin 2006), Vershynin obtained an improved result with a much simplified proof.

Other Examples of Smoothed Analysis

Smoothed analysis has been applied to analyze the performance of several practical algorithms. The following are a few examples:

- In machine learning, Blum and Dunagan (2002) proved that the perceptron algorithm usually has polynomial-time smoothed complexity. As another application of smoothed analysis in machine learning, Kalai et al. (2009) proved that all decision trees are PAC-learnable from most product distributions.
- In clustering, Arthur et al. (2009) recently settled an early conjecture of Arthur and Vassilvitskii (2006) by showing that Llyods *k*-means algorithm has polynomial smoothed complexity.
- In local search, Englert et al. (2007) showed that the smoothed complexity of the 2-Opt local search algorithm for TSP is polynomial.
- In stochastic optimization, Nikolova et al. (2006) proved that the smoothed complexity of the parametric shortest-path problem is polynomial.
- In combinatorial optimization, Röglin and Vöcking (2007) demonstrated that various integer programming problems for packing and covering with fixed number of constraints are in smoothed polynomial time.
- In linear programming, Vershynin (2006) greatly improved the analysis of Spielman and Teng and showed that the smoothed complexity of the shadow-vertex method depends only logarithmically in the number of variables.
- In mathematical programming, Kelner and Nikolova (2007) showed that the general fixed-dimensional quasi-concave minimization problem is in smoothed polynomial time.
- In multiobjective optimization with a constant number of objective functions, Röglin and Teng (2009) showed that the number of Pareto points in polynomial.

In these results, various structural properties of perturbed instances have been discovered and used. In recent years, motivated by smoothed analysis, there have been several substantial advances in analyzing the condition number of perturbed matrices. For real matrices, Sankar et al. (2005) established the following smoothed bound on the condition number:

Theorem 2 (Sankar-Spielman-Teng). *Let* $\bar{\mathbf{A}}$ *be an arbitrary square matrix in* $\mathbb{R}^{n \times n}$*, and let* **A** be a σ-Gaussian perturbation of $\bar{\mathbf{A}}$. Then

$$Pr_{\mathbf{A}}\left[||\mathbf{A}^{-1}||_2 \geq x\right] \leq 2.35 n^{1/2} (x\sigma)^{-1}.$$

Drawing from this continuous theorem and an earlier result of Edelman on random matrices, Spielman and Teng asked the following conjecture:

Conjecture 1 (Spielman-Teng). Let **A** be an n by n matrix of independently and uniformly chosen ± 1 entries. Then

$$Pr_{\mathbf{A}}\left[||\mathbf{A}^{-1}||_2 \geq x\right] \leq n^{1/2} x^{-1} + \alpha^n.$$

Moreover, for any n by n matrix $\bar{\mathbf{A}}$ of $\pm 1^{'}$s. Let **A** be a σ-Boolean perturbation of $\bar{\mathbf{A}}$. Then

$$Pr_{\mathbf{A}}\left[||\mathbf{A}^{-1}||_2 \geq x\right] \leq O(n^{1/2} (x\sigma)^{-1}).$$

Recently, Vu and Tao (2007) and Rudelson and Vershynin (2006) proved this conjecture.

Perturbation-Based Algorithm Design

In this section, we present three examples in which perturbation has played a critical role in algorithm design. All these algorithms use the following design paradigm:

Perturbation-Based Algorithm Design

1. Determine a form of instances and a set of input conditions for which efficient algorithm exists.
2. Transform the given input instance to the desired form.
3. Perturb the transformed instance so that the conditions above are satisfied.
4. Solve the perturbed instance.
5. If the solution to the perturbed instance is too far from being a solution to the transformed instance, iteratively apply the transformation/perturbation steps to close this gap.

Well-Shaped Mesh Generation

Our first example deals with three-dimensional mesh generation. To state the problem, we first introduce a few notations. Suppose $S_0 \in [0, 1)^3$ is a finite set of points. Let Z^3 be the three-dimensional integer grid. Then, $S = S_0 + Z^3$ is called the *periodic point set* generated by S_0. The consideration of periodic point sets suppress the distraction of boundary effect (Cheng et al. 2000).

The *radius-edge ratio* of a tetrahedron is the ratio of the radius of its circumsphere to the length of its shortest edge (Miller et al. 1995). The *aspect-ratio* of a tetrahedron is the ratio of the radius of its smallest containing sphere to the radius of the largest sphere contained in the tetrahedron (Cheng et al. 2000). For a constant $\rho > 1$, a periodic point set S satisfies the Ratio Property $[\rho]$, if the radius-edge ratio of every tetrahedron of the Delaunay triangulation of S is at most ρ.

Problem Statement

- **Input**: a periodic point set $S \in \mathbb{R}^3$ satisfying Ratio Property $[\rho]$ and a shape parameter τ.
- **Output**: a triangulation T of S such that the aspect-ratio of every tetrahedron in T is at least τ.

We refer to the triangulation T as a *τ-well-shaped mesh* of S.

A Key Perturbation Lemma of Weighted Points

A weighted 3D point $\hat{\mathbf{p}} = (\mathbf{p}, P) \in \mathbb{R}^3 \times \mathbb{R}^1$ is the sphere of radius P centered at $\mathbf{p}$. Traditional points can be viewed as weighted points with 0 weight. The *weighted distance* between weighted points $\hat{\mathbf{p}}$ and $\hat{\mathbf{q}} = (\mathbf{q}, Q)$ is defined as

$$||\hat{\mathbf{p}} - \hat{\mathbf{q}}|| = \left(||\mathbf{p} - \mathbf{q}||^2 - P^2 - Q^2\right)^{1/2}.$$

The weighted points $\hat{\mathbf{p}}$ and $\hat{\mathbf{q}}$ are *orthogonal* if their weighted distance is 0.

The statement that “every four points in 3D have a circumsphere” can be extended to “every four weighted points in 3D have a common orthogonal sphere.” This sphere is called their *orthosphere*. With this extension, we can easily generalize the concept of Voronoi Diagram and Delaunay triangulation of points in 3D to *weighted Voronoi Diagram* and *weighted Delaunay triangulation* of weighted 3D points. In other words, a triangulation T of a set $\hat{S}$ of weighted points is a weighted Delaunay triangulation of $\hat{S}$ if the orthosphere of every tetrahedron has a non-negative distance to each of the weighted points in $\hat{S}$.

In the context of mesh generation, Cheng et al. (2000) viewed the input point set S as a set of weighted points (with weight 0). They proved the following statement: There exists a perturbation to the weights to make the weighted Delaunay triangulation well-shaped. In particular, they proved the following key perturbation lemma about weighted Delaunay triangulation:

Lemma 2 (Cheng-Dey-Edelsbrunner-Facello-Teng). *For any $\rho > 0$, there exists a constant $\tau > 0$ depending only on ρ such that for every periodic point set S satisfying the Ratio Property [ρ], there exists a perturbation $\hat{S}$ to the weights associated with points with centers S, such that the weighted Delaunay triangulation of $\hat{S}$ is τ-well-shaped.*

We refer to the $\hat{S}$ as a *sliver-free* perturbation of S.

Perturbation-Based Algorithms and Extensions

The perturbation Lemma 2 can be utilized for efficient mesh generation.

Algorithm `SliverExudation`(S)

1. Compute a sliver-free perturbation of the weights of S and let $\hat{S}$ denote the resulting weighted points.
2. Return the weighted Delaunay triangulation of $\hat{S}$.

With some additional perturbation techniques, Cheng et al. (2000) presented two weight assignment algorithms for Step 1 above. The first one is sequential and the second one is parallel. Both of these algorithms are asymptotically optimal.

`SliverExudation` can then be used to design an efficient mesh generation algorithm for the following more general input.

- **Input**: a periodic point set $S = S_0 + Z^3$ and a shape parameter τ,
- **Output**: a small point set S'_0 and a triangulation T of $S \cap (S'_0 + Z^3)$ such that T is τ-well-shaped.

Algorithm `WellShaped3DMeshingOfPeriodPoints`(S)

1. Let $S'_0 =$ `IterativeDelaunayRefinement`(S).
2. Return `SliverExudation`($S \cup (S'_0 + Z^3)$)

`IterativeDelaunayRefinement`(S) starts with the Delaunay triangulation of S. Initially, $S'_0 = \emptyset$. Suppose the current Delaunay triangulation is T_c. If there is a tetrahedron in T_c with radius-edge ratio more than 2, then add the circumcenter of that tetrahedron to S'_0 and let T_c be the Delaunay triangulation of $S \cup (S'_0 + Z^3)$. This refinement step is repeated until all tetrahedra have radius-edge ratios of at most 2.

Shewchuck (1998) proved that `IterativeDelaunayRefinement`(S) always terminates with a point set S'_0 such that (1) $S \cup (S'_0 + Z^3)$ satisfies Ratio Property [ρ] with $\rho = 2$ and (2) $|S'_0|$ is within a constant factor needed in the best possible solution. Therefore, one can show that:

Theorem 3. `WellShaped3DMeshingOfPeriodPoints`*(S), on any periodic point set S, returns a well-shaped mesh T whose size is within a constant factor* *needed in the best possible solution. Moreover, this algorithm can be implemented to run in $O(|T| \log |T|)$ time.*

Combining the steps of `IterativeDelaunayRefinement` and `SliverExudation`, Li and Teng (2001) developed a perturbation scheme: Instead of inserting the circumcenter of the tetrahedron with radius-edge ratio more than 2 as in `IterativeDelaunayRefinement`, they find a perturbation of that circum-center to ensure that the resulting tetrahedron satisfies some additional properties. They proved the following theorem.

Theorem 4 (Li-Teng). *There is a meshing algorithm that for any periodic point set S, computes a set $S_0^{'}$ such that the Delaunay triangulation of $S \cup (S_0^{'} + Z^3)$ is well-shaped. Moreover, $|S_0^{'}|$ is within a constant factor needed in the best possible solution and the algorithm can be implemented to run in $O(|T| \log |T|)$ time.*

This perturbation-based meshing refinement method leads to an algorithm for handling arbitrary geometry boundaries (Li and Teng 2001), solving a long standing open question in 3D mesh generation. Another perturbation-based algorithm for sliver removal is given in Edelsbrunner et al. (2000).

Polynomial-Time Simplex Algorithm

Our second example is about linear programming. We will give a high-level review of the randomized simplex algorithm of Kelner and Spielman (2006).

Problem Statement

- **Input**: an $m \times n$ matrix $\mathbf{A}$, an m-place vector $\mathbf{b}$, and an n-place vector $\mathbf{c}$.
- **Output**: if $\mathbf{Ax} \leq \mathbf{b}$ is infeasible, then return "infeasible"; if $\mathbf{Ax} \leq \mathbf{b}$ is unbounded in the direction of $\mathbf{c}$, then return "unbounded", otherwise, return a vector $\mathbf{x}$ satisfying $\mathbf{Ax} \leq \mathbf{b}$ that maximizes $\mathbf{c}^T\mathbf{x}$.

A Key Perturbation Lemma of Polytopes

Let $B(\mathbf{0}, t)$ be the ball of radius t centered at the origin. A convex set is *k-round* if it contains $B(\mathbf{0}, 1)$ and is contained in $B(\mathbf{0}, k)$, where $B(\mathbf{0}, t)$ is the ball of radius t centered at the origin.

Lemma 3 (Kelner-Spielman). *Let V be the span of two uniformly random unit vectors. Let $P = \{\mathbf{x} \mid \mathbf{a}_{\mathrm{i}}^{\mathrm{T}}\mathbf{x} \leq 1, \text{for } i \in [1:m]\}$ be a k-round n-dimensional polytope defined by m inequalities. Let $Q = \{\mathbf{x} \mid \mathbf{a}_i^T\mathbf{x} \leq 1 + r_i, \text{ for } i \in [1:m]\}$ be a random perturbation of P, where r_{i} is an independent exponentially distributed random variable with expectation λ. Then, the expected shadow size defined by the projection of Q onto V is at most $O(k \max(1, \lambda \ln n) m^{1/2} n \lambda^{-1})$.*

This lemma can be strengthened to allow one of the unit vectors defining V to be a slight perturbation of a given vector. Moreover, the assumption that P is k-round can be removed if one measures the shadow size of $Q \cap B(\mathbf{0}, k)$ instead of the shadow size of Q onto V.

Perturbation-Based Simplex Algorithm

The basic idea of Kelner and Spielman (2006) is to reduce a linear programming problem in form (15.1) to the problem of deciding whether a linear program of form

$$\mathbf{Bw} \leq \mathbf{h}, \tag{15.4}$$

is bounded. This reduction can be performed in polynomial time.

The key observation is that the boundedness of program (15.4) is independent of the choice of the righthand vector $\mathbf{h}$. For example, one can simply choose $\mathbf{h} = \mathbf{1}$. To use Lemma 3 and its extension for determining the boundedness of (15.4), Kelner and Spielman choose $\mathbf{h}$ to be an exponential perturbation of $\mathbf{1}$. They then attempt to solve the boundedness problem by running the shadow-vertex simplex method: They choose a random objective function $\mathbf{c}$ and a properly perturbed starting direction to define the two-dimensional plane V for the shadow projection.

If the initial program defines a polynomially-round polytope, then the application of the shadow-vertex method would yield a randomized polynomial time algorithm. It returns a pair of vertices that optimize $\mathbf{c}$ and $-\mathbf{c}$, from which the boundedness can be certified. Kelner and Spielman cleverly use the information when the shadow-vertex method fails to determine the boundedness: They proceed as if the starting polytope is polynomially round. If the perturbation based shadow-vertex method did not find an optimal solution to $\mathbf{c}$ in polynomial steps, with high probability, it discovers a point of large norm inside the polytope. This point can be used to improve the quality of shadow-vertex search: It is well-known that for every polytope, there exists an affine transformation that makes the polytope polynomially round. Finding such a transformation is nontrivial. However, a point in the polytope with large norm can be used to produce a transformation that improves the roundness of the polytope. Kelner and Spielman use this transformation to compute a better distribution for restarting the shadow-vertex search. By carefully putting these pieces together, they give the first randomized weakly polynomial time simplex-like algorithm for linear programming.

Theorem 5 (Kelner-Spielman). *If each entry of* $\mathbf{A}$, $\mathbf{b}$, $\mathbf{c}$ *of a linear program in form* (15.1) *is specified with L bits, then the perturbation-based simplex algorithm outlined above can solve it in* $\mathrm{O}(\mathrm{n}^3 \mathrm{L} \log \delta^{-1})$ *iterations, with probability at least* $1 - \delta$.

Note that one can use standard boosting techniques to further improve the probability for finding a solution.

Robust Gaussian Elimination

Our third example is about the design of a robust algorithm for solving linear systems. Finding the solution to a linear system $\mathbf{Ax} = \mathbf{b}$ is the most fundamental problem in scientific computing (Strang 1980; Golub and Van Loan 1989; Demmel 1997). Although the classic Gaussian elimination algorithm takes $O(n^3)$ operations to solve a linear system with n variables and n equations, the precision needed can vary greatly from system to system.

Using standard stability analysis, one can show that at least

$$\max(b, \log_2 n, \log_2 \kappa(\mathbf{A}))$$

bits of accuracy are needed to obtain a solution that is accurate to b bits, where $\kappa(\mathbf{A})$ is the condition number of $\mathbf{A}$. Note that $\kappa(\mathbf{A}) = ||\mathbf{A}||_2 ||\mathbf{A}^{-1}||_2$, where $||\mathbf{A}||_2 = \max_{\mathbf{x}} ||\mathbf{Ax}||_2 / ||\mathbf{x}||_2$, and $||\mathbf{A}^{-1}||_2$ is also known as the smallest singular value of $\mathbf{A}$. Wilkinson (1961) constructed a family of counterexamples showing that in the worst-case one must use at least $\Omega(n)$ bits to accurately solve every linear system with the Gaussian elimination algorithm that uses the partial pivoting rule.

Problem Statement

- **Input**: a linear system $\mathbf{Ax} = \mathbf{b}$ and an integer b.
- **Output**: a solution to the system that is accurate to b bits.
- **Objective**: design an algorithm that uses precision linearly in b, $\log_2 n$, and $\log_2 \kappa(\mathbf{A})$.

A Perturbation Lemma

Gaussian elimination systematically reduces the input system to ones that have smaller number of variables: At each step, it chooses one of the equations and one of the variables, and uses the chosen equation to eliminate the variable from other equations. Eventually, it either concludes that the system has no solution, multiple solutions, or exactly one solution. In the last case, the elimination process obtains a single linear system with only one variable. It then solves that system, and uses backward substitution to find the solution values of other variables that have been eliminated. It solves the linear system in the second case similarly.

The choice of the equation and the variable is determined by a pivoting rule. The simplest pivoting rule is to use the i-th equation to eliminate the i-th variable. But the pivoting rule commonly used in practice is *partial pivoting*. In the i-th step, it chooses the equation in which the i-th variable has the largest coefficient of absolute value, and uses that equation to eliminate the i-th variable.

Gaussian elimination with partial pivoting defines a row-permutation matrix **P** and factors **PA** into

$$\mathbf{PA} = \mathbf{LU}.$$

Because of the partial pivoting, all entries in **L** have absolute value of at most 1.

In his seminal work (Wilkinson 1961), Wilkinson considered the number of bits needed to obtain a solution with a given accuracy. He proved that it suffices to carry out Gaussian elimination with

$$b + \log_2(5n\kappa(\mathbf{A})||\mathbf{L}||_\infty||\mathbf{U}||_\infty/||\mathbf{A}||_\infty + 3)$$

bits of accuracy to obtain a solution that is accurate to b bits. In the formula, $||\mathbf{A}||_\infty$ is the maximum absolute row sum. The quantity $||\mathbf{L}||_\infty||\mathbf{U}||_\infty||\mathbf{A}||_\infty$ is called the *growth factor* of the elimination. It depends on the pivoting rule.

In (Sankar et al. 2005), Sankar et al. established the following smoothed bound:

Lemma 4 (Sankar-Spielman-Teng). *For $n > e^4$, let $\bar{\mathbf{A}}$ be an n-by-n matrix for which $||\bar{\mathbf{A}}||_2 \leq 1$, and let $\mathbf{A}$ be a σ-Gaussian perturbation of $\bar{\mathbf{A}}$, for $\sigma \leq 1/2$. Then,*

$$E[\log \rho_{GEWP}(\mathbf{A})] \leq 3\log_2 n + 2.5\log_2\sigma^{-1} + 0.5\log_2\log_2 n + 1.81,$$

where ρ_{GEWP} (A) is the growth factor of Gaussian elimination without pivoting.

A Robust Algorithm for Linear System

As Wilkinson pointed out, in the worst case, one needs $\log\kappa(\mathbf{A})$ digits of precision to solve a linear system, as large errors occur if one uses any fewer bits to store A or **b**. Sankar, Spielman and Teng observed that their perturbation lemma above leads to an algorithm for solving linear systems whose precision only depends upon $\kappa(\mathbf{A})$. The algorithm is the following: perturb **A** by a Gaussian of norm at most $O((n^{1/2}\kappa(A))^{-1})$, then solve the system by Gaussian elimination without pivoting. It is easy to demonstrate that the solution to this perturbed system is an approximation of the solution of the original. Moreover, Lemma 4 implies that the elimination can be performed with low precision, with high probability.

Theorem 6 (Sankar-Spielman-Teng). *Let* A *be an n-by-n matrix and let* b *be a vector. If we perturb* **A** *by adding a Gaussian random matrix of standard deviation $\delta(2^{b+3}n^{1/2}\kappa(A))^{-1}$ and solve the perturbed system using Gaussian elimination without pivoting and*

$$4b + 10\log(n) + 3\log(\kappa(A)) + 5\log(1/\delta) + 7$$

bits of precision, then with probability at least $1 - \delta$ the solution we obtain is a solution for the original system that is accurate to b bits.

Note that one can use standard boosting techniques to further improve the chance of getting a correct solution in the above theorem without increasing the required precision. To do so, we apply our robust solver multiple times and return the most accurate solution. For example, by running our robust solver twice, we can improve the probability of success from $1 - \delta$ to $1 - \delta^2$.

We also note that this bound is only slightly off from the lower bound of log $(\kappa(\mathbf{A})) + b$ that trivially holds for every algorithm. Although it remains open theoretically, in practice, we can apply Gaussian elimination with partial pivoting to achieve better precision.

Other Algorithmic Applications of Smoothed Analysis

In a recent work, Dughmi and Roughgarden (Dughmi and Roughgarden 2010) gave a black-box reduction in algorithmic mechanism design for the class of packing problems. They proved that if packing problem has a fully polynomial time approximation scheme, then it also admits a truthful-in-expectation randomized mechanism that is also a fully polynomial time approximation scheme. Their reduction uses perturbation as a tool to achieve truthfulness. In their proof, they applied the Röglin and Teng (2009) characterization of the polynomial smoothed complexity for binary packing problems.

Smoothed Complexity Versus Approximation Complexity

Perturbations have been used to design approximation algorithms. For example, the Euclidean TSP approximation algorithms of Arora (1998) and Mitchell (1999) first perturb each input point to its closest grid point of a chosen scale, and then use the optimal tour for the perturbed points as the approximation solution.

The connection between the smoothed complexity and approximation complexity can be exploited in the complexity study of both measures. Below, I use the two-player Nash equilibrium as our example.

A *two-player game* (Nash 1951; Lemke 1965; Lemke and Howson 1964) can be specified by a pair of $m \times n$ payoff matrices $(\mathbf{A}, \mathbf{B})$. Without loss of generality, we can assume that all payoff entries are between 0 and 1.

Let $\mathbb{P}^n$ denote the set of all *probability vectors* in $\mathbb{R}^n$, i.e., non-negative, n-place vectors whose entries sum to 1. Then, two column vectors $(\mathbf{x}^* \in \mathbb{P}^m, \mathbf{y}^* \in \mathbb{R}^n)$ is a Nash equilibrium of $(\mathbf{A}, \mathbf{B})$ if for all $\mathbf{x} \in \mathbb{R}^m$ and $\mathbf{y} \in \mathbb{R}^n$:

$$(\mathbf{x}^*)^T \mathbf{A} \mathbf{y}^* \geq \mathbf{x}^T \mathbf{A} \mathbf{y}^* \quad \textit{and} \quad (\mathbf{x}^*)^T \mathbf{B} \mathbf{y}^* \geq (\mathbf{x}^*)^T \mathbf{B} \mathbf{y}.$$

For a positive parameter ε, an *ε-approximate Nash equilibrium* of a two-player game $(\mathbf{A}, \mathbf{B})$ is a pair $(\mathbf{x}^* \in \mathbb{R}^m, \mathbf{y}^* \in \mathbb{R}^n)$ such that for all $\mathbf{x} \in \mathbb{R}^m$ and $\mathbf{y} \in \mathbb{R}^n$:

$$(\mathbf{x}^*)^T \mathbf{A}\mathbf{y}^* \geq \mathbf{x}^T \mathbf{A}\mathbf{y}^* - \epsilon \ \ \textit{and} \ \ (\mathbf{x}^*)^T \mathbf{B}\mathbf{y}^* \geq (\mathbf{x}^*)^T \mathbf{B}\mathbf{y} - \epsilon.$$

In the smoothed analysis of the two-player game, we consider perturbed games in which each entry of the payoff matrices is subject to a small and independent random perturbation. For simplicity, we consider the *uniform perturbation* model. Suppose $\overline{\mathbf{A}} = (\bar{a}_{i,j})$ and $\overline{\mathbf{B}} = (\bar{b}_{i,j})$ are two matrices of the same size. For a magnitude parameter σ, a perturbed instance $(\mathbf{A}, \mathbf{B})$ is obtained from the two-player game $(\overline{\mathbf{A}}, \overline{\mathbf{B}})$ by replacing the payoff entries $\bar{a}_{i,j}$ and $\bar{b}_{i,j}$, by a value chosen uniformly at random from $[\bar{a}_{i,j} - \sigma, \bar{a}_{i,j} + \sigma]$ and from $[\bar{b}_{i,j} - \sigma, \bar{b}_{i,j} + \sigma]$, respectively.

The following lemma connects the smoothed complexity of two-player Nash equilibrium with its approximation complexity.

Lemma 5 (Smoothed Nash vs Approximate Nash). *If there is an algorithm with polynomial smoothed complexity for finding a two-player Nash equilibrium, then for all* $\varepsilon > 0$*, there exists a randomized algorithm for computing an ε-approximate Nash equilibrium in a two-player game with expected running time* $\mathrm{O}\left(\mathrm{poly}(\mathrm{m}, \mathrm{n}, 1/\varepsilon)\right)$.

In (Chen et al. 2009a), Chen et al. proved the following theorem:

Theorem 7 (Chen-Deng-Teng). *If there is an algorithm that computes an ε-approximate Nash equilibrium of a two-player game in time* $\mathrm{O}(\mathrm{poly}(\mathrm{m}, \mathrm{n}, 1/\varepsilon))$*, then every problem in the complexity class* **PPAD** *is solvable in polynomial time.*

Consequently, using the connection between the smoothed complexity and approximation complexity of Lemma 5, one can show:

Theorem 8 (Smoothed Nash). *The problem of finding a Nash equilibrium of a two-player game is not in smoothed polynomial time unless* **PPAD** *is contained in* **RP**.

The smoothed complexity and the approximation results above have also been extended to the computation of Arrow-Debreu equilibrium prices in exchange markets (Huang and Teng 2007; Chen et al. 2009b).

The Laplacian Paradigm: Emerging Algorithms for Massive Graphs

In this section, we present our second example of numerical thinking. In this example, graph algorithms are designed for solving a numerical problem. In the process, numerical consideration has led to the introduction of several new graph-theoretic concepts. The resulting numerical algorithms have become the basis of a new algorithmic framework.

Nearly-Linear Time Laplacian Primitive

A matrix $\mathbf{L} = (l_{i,j})$ is a *Laplacian matrix* if (1) it is symmetric, i.e., $l_{i,j} = l_{j,i}$ for all i,j, (2) $l_{i,j} \leq 0$ for all $i \neq j$, and (3) $l_{i,i} = -\sum_{j \neq i} l_{i,j}$ for all i. We can view an $n \times n$ Laplacian matrix as a weighted undirected graph over n vertices.

Let $G = (V, E)$ be a graph with n vertices $V = \{\ 1, \ldots, n\}$. The *adjacency matrix*, $A(G)$, of a graph $G = (V, E)$ is the $n \times n$ matrix whose (i,j)-th entry is 1 if $(i,j) \in E$ and 0 otherwise, and the diagonal entries are defined to be 0. Let D be the $n \times n$ *diagonal matrix* with entries $D_{i,i} = d_i$, where d_i is the degree of the ith vertex of G. The *Laplacian*, $L(G)$, of the graph G is defined to be $L(G) = D - A$.

In general, suppose $G = (V, E, w)$ is a weighted undirected n-vertex graph where each edge in $e \in E$ has a weight $w(e) > 0$ and for each $e \notin E$, $w(e) = 0$. Sometime we say w defines the *affinity* between each pair of vertices. We can extend the notion of adjacency matrix $A(G)$, diagonal matrix $D(G)$ and Laplacian matrix $L(G)$ to weighted graphs as following: $A_{i,j}(G) = w(i,j)$ and $D_{i,i}(G) = \sum_{j \neq i} w(i,j)$ and $L(G) = D(G) - A(G)$.

The Laplacian Primitive and its Solver

A fundamental problem in numerical analysis is to find a solution to a linear system. Mathematically, we are given an $n \times n$ matrix $\mathbf{A}$ and an n-place vector $\mathbf{b}$ (in the column span of $\mathbf{A}$), and are asked to find a vector $\mathbf{x}$ such that $A\mathbf{x} = \mathbf{b}$. In practice, we are often allowed to have a small degree of imprecision. For example, given a precision parameter ε, we are asked to produce an $\tilde{\mathbf{x}}$ such that $\|\tilde{\mathbf{x}} - \mathbf{A}^{\dagger}\mathbf{b}\|_2 \leq \epsilon \|\mathbf{A}^{\dagger}\mathbf{b}\|_2$, where $\mathbf{A}^{\dagger}$ denotes the Moore-Penrose pseudo-inverse of $\mathbf{A}$ – that is the matrix with the same nullspace as $\mathbf{A}$ that acts as the inverse of $\mathbf{A}$ on its image.

We will call the computational problem of solving a linear system defined by a Laplacian matrix the *Laplacian Primitive*.

Definition 3 (Laplacian Primitive). *This primitive concerns linear systems defined by Laplacian matrices*:

> `Input`: *a Laplacian matrix* $\mathbf{L}$ of dimension n, an n-place vector $\mathbf{b} = (\mathbf{b}_1, \ldots, \mathbf{b}_n)^T$ **such that** $\sum_i \mathbf{b}_i = 0$**, and a precision parameter** $\boldsymbol{\varepsilon} > 0$.
>
> `Output`: *an n-place vector* $\tilde{\mathbf{x}}$ *such that* $\|\tilde{\mathbf{x}} - \mathbf{L}^{\dagger}\mathbf{b}\|_L \leq \epsilon \|\mathbf{L}^{\dagger}\mathbf{b}\|_{\mathbf{L}}$, *where for an n-place vector* $\mathbf{z}$, its $\mathbf{L}$ norm *is defined as* $\mathbf{z}^T\mathbf{L}\mathbf{z}$.

The starting point of the Laplacian Paradigm to be discussed in the next section is the following algorithmic result for solving Laplacian linear systems (Spielman and Teng 2008b).

Theorem 9 (Spielman-Teng). *There is a randomized algorithm for the Laplacian primitive that runs in expected time* $m \log^{O(1)} n \log(1/\varepsilon)$*, where n is the dimension of the Laplacian matrix, m is the number of non-zero entries in the Laplacian matrix, and ε is the precision parameter.*

Note that this result makes no assumption on the structure of the non-zero entries. In fact, the solver of Spielman-Teng applies to every linear system $\mathbf{Ax} = \mathbf{b}$ where $\mathbf{A}$ is a symmetric, weakly diagonally dominant matrix. A matrix is *weakly diagonally dominant* if the diagonal entry of each row is at least the 1-norm of the off-diagonal entries of that row.

The Laplacian solver applies the combinatorial preconditioning technique introduced in the pioneering work of Vaidya (1991). It also uses insights and results in the work of Joshi, Reif, Gremban, Miller, Boman, Hendrickson, Maggs, Parekh, Ravi, Woo, Bern, Gilbert, Chen, Nguyen, Toledo (Boman and Hendrickson 2003; Bern et al. 2006; Joshi 1997; Rief 1998; Gremban 1996; Maggs et al. 2005).

A Suite of Nearly-Linear-Time Spectral Algorithms

In the process of developing the nearly linear-time algorithm for the Laplacian primitive, Spielman and Teng and their collaborators designed a suite of nearly linear-time graph algorithms. Most of these algorithms concern the spectral property of graphs. We include these nearly linear-time spectral algorithms as part of the algorithmic primitives in the Laplacian Paradigm.

Clustering and Partitioning

The first family of their spectral algorithms is for clustering and partitioning. A *cluster* of $G = (V, E, w)$ is a subset of V that is richly intra-connected but sparsely connected with the rest of the graph. The quality of a cluster can be measured by its *conductance*, the ratio of the number of its external connections to the number of its total connections.

We let $d(i) = D_{i,i}(G)$ denote the *weighted degree* of vertex i. For $S \subseteq V$, we define $\mu(S) = \sum_{i \in S} d(i)$. So, $\mu(V) = 2\,|\,E\,|$ if the weights of all edges are equal to 1. Let $E(S, V - S)$ be the set of edges connecting a vertex in S with a vertex in $V - S$. We define the *conductance* of a set of vertices S, written $\Phi(S)$, and the *conductance* of G, respectively by

$$\Phi(S) \overset{\text{def}}{=} \frac{|E(S, V - S)|}{\min(\mu(S), \mu(V - S))}, \quad \text{and} \quad \Phi_G \overset{\text{def}}{=} \min_{S \subset V} \Phi(S).$$

We also refer to a subset S of V as a *cut* of G and refer to $(S, V - S)$ as a *partition* of G. The *balance* of a cut S or a partition $(S, V - S)$ is then equal to $bal(S) = \min(\mu(S), \mu(V - S))/\mu(V)$. We call S a *sparsest cut* of G if $\Phi(S) = \Phi_G$ and $\mu(S)/\mu(V) \leq 1/2$.

The clustering problem has centered around the following combinatorial optimization problem: Given an undirected graph G and a conductance parameter, find a cluster C such that $\Phi(C) \leq \phi$, or determine no such cluster exists. The problem is

NP-complete (Leighton and Rao 1999). But, approximation algorithms exist. Leighton and Rao (1999) used linear programming to obtain $O(\log n)$-approximations of the sparsest cut. Arora et al. (2004) improved this to $O(\sqrt{\log n}\,)$ through semi-definite programming. Subsequently, faster algorithms obtaining similar guarantees have been constructed (Arora et al. 2004; Khandekar et al. 2006; Arora and Kale 2007; Orecchia et al. 2008).

The algorithmic kernel of the Laplacian solver of Spielman and Teng is a local-clustering algorithm, called `Nibble`, for weighted graphs, based on random walk distributions (Spielman and Teng 2008a). The running time of this algorithm is almost linear in the size of the cluster it produces, and is almost independent of the size of the original graph. Although the algorithm may not find a local cluster for some input vertices, it is usually successful:

Theorem 10 (Local Clustering). *There exists a constant $\alpha > 0$ such that for any target conductance ϕ and any cluster C_0 of conductance at most $\alpha \cdot \phi^2/\log^3 n$, when given a random vertex v sampled according to degree inside C_0,* `Nibble` *will return a cluster C mostly inside C_0 and with conductance at most ϕ, with probability at least 1/2.*

Using `Nibble` as a subroutine, Spielman and Teng (2008a) developed an algorithm called `Partition` and prove the following statement.

Theorem 11 (Nearly Linear-Time Partitioning). *There exists a constant $\alpha >$ 0 such that for any graph $G = (V,E)$ that has a cut S of sparsity $\alpha \cdot \theta^2/\log^3 n$ and balance $b \leq 1/2$, with high probability,* `Partition` *finds a cut D with $\Phi_V(D) \leq \theta$ and* ***bal***(D) $\geq$ b/2.

Spectral Graph Sparsification

One of the major conceptual developments in the work of Spielman and Teng (2003, 2008c) is a new notion of graph sparsification based on the spectral similarity of graph Laplacians. Let **L** be an $n \times n$ Laplacian matrix. An n-dimensional vector $\mathbf{x} = (x_1, \ldots, x_n)^T$ is an *eigenvector* of **L** if there is a scalar λ such that $\mathbf{L}\mathbf{x} = \lambda\mathbf{x}$. λ is the *eigenvalue* of **L** corresponding to the eigenvector **x**. Because **L** is a symmetric matrix, all of its n eigenvalues are real. Notice that the all-1′s vector is an eigenvector of any Laplacian matrix and its associated eigenvalue is 0. Because Laplacian matrices are positive semidefinite, all the other eigenvalues must be non-negative. An important property of weighted Laplacian is: for all $\mathbf{x} \in \mathbb{R}^{|V|}$

$$\mathbf{x}^T\mathbf{L}\mathbf{x} = \sum_{i,j} l_{i,j}(x_i - x_j)^2.$$

Graph sparsification is the task of approximating a graph by a sparse graph, and is often useful in the design of efficient approximation algorithms. Several notions of graph sparsification have been proposed. For example, Chew (1986) was motivated by proximity problems in computational geometry to introduce graph spanners. Spanners are defined in terms of the distance similarity of two graphs: A spanner is a sparse graph where the shortest-path distance between every pair of vertices is approximately the same in the original graph as in the sparsifier. Motivated by cut problems, Benczur and Karger, introduced a notion of sparsification that requires that for every set of vertices, the weight of the edges leaving that set should be approximately the same in the original graph as in the sparsifier.

Motivated by constructing preconditioners, Spielman and Teng introduce a new notion of sparsification called *spectral sparsification* (Spielman and Teng 2008c). A *spectral sparsifier* is a subgraph of the original whose Laplacian quadratic form is approximately the same as that of the original graph on all real vector inputs. We say that $\widetilde{G}$ is a σ*-spectral approximation* of G if for all $\mathbf{x} \in \mathbb{R}^V$

$$\frac{1}{\sigma}\mathbf{x}^T L(\widetilde{G})x \le \mathbf{x}^T L(G)x \le \sigma \mathbf{x}^T L(\widetilde{G})\mathbf{x}. \qquad (15.5)$$

This notion of sparsification captures the *spectral similarity* between a graph and its sparsifiers. It is a stronger notion than the cut sparsification of Benczur and Karger: the cut-sparsifiers constructed by Benczur and Karger are only required to satisfy these inequalities for all $\mathbf{x} \in 0, 1^V$.

In (Spielman and Teng 2008c), Spielman and Teng proved the following theorem about spectral sparsification with a nearly-linear-time algorithm:

Theorem 12 (Spectral Sparsification). *Given*$\epsilon \in (1/n, 1/3)$*, $p \in$ (0,1/2) and a weighted graph G and with n vertices, in expected time* $m\log$ *(1/p)*$\log^{O(1)}$*n, one can produce a weighted graph* $\widetilde{G}$ *that satisfies the following properties:*

(a) The edges of $\widetilde{G}$ *are a subset of the edges of G; and*
(b) With probability at least 1 − p, (b.1) $\widetilde{G}$ *is a (1 + ε)-spectral approximation of G, and (b.2)* $\widetilde{G}$ *has at most* $\epsilon^{-2} n \log^{O(1)}(n/p)$ *edges.*

Low Stretch Spanning Trees

An important discrete mathematical concept in building preconditioners is the low-stretch spanning tree introduced by Alon et al. (1995): Suppose T is a spanning tree of $G = (V, E, w)$. For any edge $e \in E$, let $e_1, \ldots, e_k \in F$ be the edges on the unique path in T connecting the endpoints of e. The *stretch* of e w.r.t T is given by $\mathrm{st}_T(e) = w(e)(\sum_{i=1}^{k} \frac{1}{w(e_i)})$. The *average stretch* of the graph G with respect to T is defined by $\mathrm{st}_T(G) = \sum_{e \in E} \mathrm{st}_T(e)/|E|$. Alon et al. proved that every weighted graph has a spanning tree with average stretch $O(n^{o(1)})$. Elkin et al. (2008), improved the average stretch to $O(\log^2 n \log\log n)$ with a nearly linear-time construction.

The Laplacian Paradigm for Massive Graphs

As an algorithmic primitive, the nearly-linear time Laplacian solver and its supporting algorithms provide a set of new tools for algorithm design. To motivate the Laplacian paradigm, we first discuss the need for nearly linear time algorithms for solving problems that involve massive graphs and data.

Massive Data and Efficient Algorithm Design

In light of the explosive growth in the amount of data and the diversity of computing applications, efficient algorithms are now needed more than ever. We may need to deal with equations and mathematical programming that involve hundreds of millions of variables (Sharma et al. 2002). We may need to analyze data and graphs such as web logs, social networks, and web graphs that are massive (e.g., of hundreds billions of nodes Gulli and Signorini 2005), complex, and dynamic. As a result of this rapid growth in problem sizes, what used to be considered an efficient algorithm, such as an $O(n^{1.5})$-time algorithm, may no longer be adequate for solving problems of these scales. Space complexity poses an even greater problem. Thus, the need to design algorithms whose running time is linear or nearly linear in the input size has become increasingly critical.

Over the last half century, several algorithmic paradigms have been developed and applied to various problems and applications. Some of these paradigms such as divide-and-conquer, dynamic programming, greedy and local search, linear and convex programming, randomization, and branch-and-bound are commonly covered in textbooks on algorithm design and analysis (Cormen et al.) while some less theoretically-covered paradigms such as the multilevel method, simulated annealing, and the genetic algorithm, are also widely used in practice.

Algorithms produced by paradigms such as dynamic programming, linear/convex programming, and branch-and-bound have running time that is typically quadratic, cubic, or of even higher order in the input size. But the algorithmic paradigms such as greedy and divide-and-conquer, when they can be successfully applied, usually lead to linear- or nearly-linear-time algorithms. In graph theory, several previously-known divide-and-conquer algorithms, run in nearly linear time or use only linear space (Frieze et al. 1992; Lipton et al. 1979). Their success critically uses the fact that the underlying graphs have a balanced separator that can be found in linear time. Thus, these algorithms can only be applied to special families of graphs, for example planar graphs (Lipton et al. 1979) and nearest neighborhood graphs (Miller et al. 1997). However, most graphs such as web graphs and social network graphs simply do not have balanced separators of the required quality.

While paradigms such as the multilevel method usually lead to nearly linear-time algorithms in practice, their theoretical behaviors remain widely open and are subjects for excellent research projects.

Many basic graph-theoretic problems such as connectivity and topological sorting can be solved in linear or nearly-linear time. The efficient algorithms for these problems are built on traditional linear-time primitives such as breadth-first-search (BFS) and depth-first-search (DFS). Minimum Spanning Trees (MST), Shortest-Path Trees, and sorting. However, many graph problems can not be directly reduced to these primitives in linear or nearly linear time.

The Laplacian Paradigm

The thesis behind the Laplacian Paradigm is that the Laplacian primitive, which was not available for previous algorithmic paradigms for graphs, could be a very powerful primitive for combinatorial optimization and numerical analysis. Unlike the separator-based divide-and-conquer paradigm, this primitive makes no assumption about the structure of the graph. Its complexity depends only (nearly) linearly on the number of vertices and edges in the underlying graph. Moreover, its complexity is logarithmic in the reciprocal of the desired precision.

We anticipate that more graph-theoretical problems can be solved in nearly-linear time using this primitive.

Schematically, to apply the Laplacian Paradigm to solve a problem defined on a graph $G=(V,E,w)$ or a matrix $\mathbf{A}$, we reduce the computational and optimization problem to one or more linear algebraic or spectral graph-theoretic problems whose matrices are Laplacian or Laplacian-like. The nearly-linear-time Laplacian primitive or its supporting primitives discussed in section "A Suite of Nearly-Linear-Time Spectral Algorithms" is then used to solve these algebraic and spectral problems.

Similar to other algorithmic paradigms, the details of the reduction and resulting algorithms depend on the structure of the problems that we need to solve. We now give three examples of Laplacian Paradigm.

Example I: Spectral Approximation

Our first example is to approximate the Fiedler value of a weighted graph. Recall that the Fielder value of a weighted graph $G=(V,E,w)$ is the second smallest eigenvalue of $L(G)$.

Definition 4 (Approximate Fiedler Vector and Fiedler Value). *For a Laplacian matrix* L, v is an ε-approximate Fiedler vector if **v** is orthogonal to the all-1′s vector and

$$\lambda_2(\mathbf{L}) \leq \lambda(\mathbf{v}) = \frac{\mathbf{v}^T\mathbf{L}\mathbf{v}}{\mathbf{v}^T\mathbf{v}} \leq (1+\epsilon)\lambda_2(\mathbf{L}),$$

where λ_2(L) is the Fiedler value of the graph of L.

To compute an approximate Fiedler vector, we use the classic inverse power method. Assume the eigenvalues of $\mathbf{L}$, from the smallest to the largest, are $\lambda_1 = 0$, $\lambda_2, \ldots, \lambda_n$. Let $\mathbf{v}_i$ be the eigenvector of λ_i. Note $\mathbf{v}_1$ is the all-1′s vector.

We choose a unit random vector $\mathbf{r}$ such that $\mathbf{v}_1{}^T\mathbf{r} = 0$. We can write $\mathbf{r}$ as $\mathbf{r} = \sum_{i=2}^{n} c_i \mathbf{v}_i$. Note that $\mathbf{L}^{\dagger}\mathbf{r} = \sum_{i=2}^{n} c_i \lambda_i^{-1} \mathbf{v}_i$. In general, for positive integer $t \geq 1$, $(\mathbf{L}^{\dagger})^t\,\mathbf{r} = \sum_{i=2}^{n} c_i \lambda_i^{-t} \mathbf{v}_i$. Therefore, if c_2 is not too small, by choosing $t = \Theta(\log(n/\epsilon)/\epsilon)$, assuming we can compute $\mathbf{L}^{\dagger}\mathbf{r}$ efficiently, we can compute an ε-approximate Fiedler vector using the inverse power method.

We can use the nearly-linear-time Laplacian primitive to approximate $\mathbf{L}^{\dagger}\mathbf{r}$ to a desired precision. With some standard techniques from numerical analysis, one can bound the approximation factor of $(\mathbf{L}^{\dagger})^t\mathbf{r}$.

Theorem 13 (Spielman-Teng). *For any ε > 0 and Laplacian matrix* $\mathbf{L}$, an ε-approximate Fiedler vector of *L can be computed by a randomized algorithm in time* $m \log^{O(1)} n \log(1/\epsilon)/\epsilon$.

It follows from Mihail (Spielman and Teng 1996) that if a graph $G(V, E)$ has a constant maximum degree, then one can obtain a cut of conductance $O((\lambda_2(G))^{1/2})$ from any approximate Fiedler vector, as guaranteed to exist by Cheeger's isoperimetric inequality (Cheeger 1970).

Corollary 1 (Cheeger Cut). *If G is a constant-degree graph of n vertices with Fiedler value* λ_2*, then in nearly linear-time, we can compute a cut of conductance* $O(\lambda_2{}^{1/2})$.

Example II: Learning from Labeled Data on a Directed Graph

Our next example is from Zhou et al. (2005). The problem is to learn from labeled and unlabeled data on a graph: The input of the problem is a strongly connected (aperiodic) directed graph $G = (V, E)$ and a labeling function y. The function y assigns a label from a label set $Y = \{1, -1\}$ to each vertex of a subset $S \subset V$ and it assignes 0 to vertices in $V - S$. Let $\mathcal{H}(V)$ be the set of functions of form $V \rightarrow \mathbb{R}$ for labeling vertices in the graph. The mathematical goal of this learning problem is to find a function $f \in \mathcal{H}(V)$ that optimizes the following objective function

$$\text{minimize } \Omega(f) + \mu||f - y||^2, \tag{15.6}$$

where μ is a constant parameter, and

$$\Omega(f) = 0.5 \sum_{(u,v) \in E} \pi(u) p(u, v) \left(f(u)\pi(u)^{-1/2} - f(v)\pi(v)^{-1/2}\right), \tag{15.7}$$

and π() is the stationary distribution of the random walk on the graph with the transition probability function $p : V \times V \rightarrow \mathbb{R}^+$ defined by the following formula:

for each pair $u, \upsilon \in V$, if $(u, \upsilon) \notin E$, then $p(u, \upsilon) = 0$; otherwise $p(u, \upsilon) = 1/d^+(u)$ where $d^+(u)$ is the out-degree of u.

Zhou, Huang, and Schölkopf proved that the optimal solution f^* to the mathematical programming defined by (15.6) is the solution to the following linear system.

$$((2+2\mu)\Pi - (\Pi P + P^T\Pi))(\Pi^{-1/2}f^*) = 2\mu\Pi^{1/2}y, \tag{15.8}$$

where Π the diagonal matrix with $\Pi(\upsilon, \upsilon) = \pi(\upsilon)$ and P is the transition probability matrix defined by $p()$. Using the property that the matrix

$$\mathbf{A} = ((2+2\mu)\Pi - (\Pi P + P^T\Pi))$$

is symmetric and diagonally dominant, Zhou, Huang, and Schölkopf applied the nearly-linear-time Laplacian primitive to obtain the following result.

Theorem 14 (Zhou-Huang-Schölkopf). *There exists a randomized algorithm that can solve the graph learning problem given by the mathematical programming defined by* (15.6) *in nearly linear time.*

Example III: Faster Maximum Flow Approximation

One of the exciting developments in Laplacian Paradigm is the recent work by Christiano et al. on maximum flow approximation. In this work, the nearly linear-time Laplacian solver is instrumental to the new flow algorithm that improves the bound achieved by the classic flow algorithm of Even and Tarjan (1975).

Recall that in the *maximum flow problem*, we are given a graph $G = (V, E, c, s, t)$ where $s \in V$ (the *source*) and $t \in V$ (the *sink*) are two special vertices in G, and $c : E \rightarrow \mathbb{R} \cup \{0\}$ assigns a *capacity* to each edge. A *feasible s-t flow* of value F from s to t is a map $f : E \rightarrow \mathbb{R}$ that satisfies

- For all $e \in E, f(e) \in [-c(e), c(e)]$,
- For each vertex $\upsilon \in V / \{s, t\}$, the sum of the flows of the edges incident to υ is 0, and
- The sum of the flows of the edges incident to s is F and the sum of the flows of the edges incident to t is $-F$.

The goal of the maximum flow problem is to compute a feasible *s-t* flow with the maximum flow value.

Even and Tarjan's algorithm works on an undirected graph where every edge has capacity 1. They showed if $m = O(n)$, then the running time of their algorithm is $O(n^{3/2})$. Even and Tarjan's result was extended by Goldberg and Rao (1998) who showed that the maximum flow problem for every directed, capacitated graph with m edges and n vertices can be computed in $O(m^{3/2})$ time. Using sparsification, Banczúr and Karger reduced the complexity to $O(mn^{1/2}\epsilon^{-1})$ to produce a flow whose value is within (1 – ε) factor of the value of the maximum flow.

Conceptually, the algorithm of Christiano et al. (CKMST algorithm) is quite simple. They reduced the maximum flow problem to the computation of many electrical flows. Suppose each edge *e* is a resistor of resistance *r*(*e*). It is well known that the electrical flow is a potential flow, that is, each node υ in the network has a potential value ϕ_υ such that $\phi_s = 1$ and $\phi_t = -1$, and for each edge (u, υ) with resistence $r_{u,\upsilon}$, the electrical flow $f(u, \upsilon)$ along the edge (u, υ) satisfies $f(u, \upsilon) = (\phi_u - \phi_\upsilon)/r_{u,\upsilon}$.

It is also well known that the vector representing these potentials is a solution to a linear system,

$$\mathbf{L} \cdot \phi = \chi_{s,t},$$

where **L** is the *n* by *n* Laplacian matrix defined by $\{1/r_e : e \in E\}$ and $\chi_{\mathrm{s,t}}$ is the vector where the entries for *s* and *t* are 1 and – 1, respectively, and all other entries are 0. Therefore, the electrical flow can be computed in nearly linear time using the Laplacian solver.

The CKMST flow algorithm views each edge of the input graph as a resistor with a proper initial resistance. It repeatedly computes the electrical flow from *s* to *t*. The electrical flow obeys the flow conservation constraints, but may not respect the capacities of the edges. To remedy this, the algorithm modifies the resistance of each edge in proportion to the amount of current flowing through it – thereby penalizing edges that violate their capacities – and computes the electrical flow with these new resistances.

Christiano et al. showed that after repeating about $O(n^{1/3} \cdot poly(1/\epsilon))$ times, the CKMST algorithm obtains a (1 – ε)-approximately maximum *s*-*t* flow by taking a certain average of the electrical flows that the iterative process has computed. Thus, this new algorithm has running time nearly $O(mn^{1/3} \cdot \ poly(1/\epsilon))$, breaking the $O(n^{3/2})$ complexity barrier for maximum flows since the 1975 work of Even and Tarjan (1975).

Other Applications of the Laplacian Paradigm

In addition to the three examples given above, the Laplacian paradigm has already been used in several problems in combinatorial optimization and scientific computing.

Boman et al. showed that the Laplacian primitive can be used to solve elliptic finite-element systems in nearly linear time. Shklarski and Toledo (2008) and Daitch and Spielman extended the solver to systems involving rigidity. Koutis et al. (2009) presented several applications of the Laplacian Paradigm in vision and image processing. Using the Laplacian primitive. Spielman and Srivastava (2008) developed a beautiful nearly linear time algorithm to compute the effective resistances in a weighted graph. Using the nearly linear-time Laplacian primitive, Madry and Kelner (2009) greatly improved the algorithm for the generation of random spanning trees; Daitch and Spielman (2008) gave the fastest known algorithm for computing generalized lossy flows; and Ding et al. (2011) gave a nearly linear time algorithm for approximating the cover times in a graph.

Next Generation Algorithms for Massive Graphs

Algorithm design is like building a software library. Once we can solve a new problem in linear or nearly linear time, we can add them to our library of efficient algorithms and use them as a subroutine in designing the next wave of algorithms. Due to the appealing property of the Laplacian primitive, as well as our algorithms for clustering, partitioning, and sparsification, we are very excited about the new possibilities of progress in algorithm design.

To support our thesis that the Laplacian Paradigm may lead to breakthrough in graph algorithms, we would like to review the previous linear solvers and their complexity. The straightforward implementation of Gaussian elimination takes $O(n^3)$ time. When m is large relative to n and the matrix is arbitrary, the fastest algorithms for solving linear equations are those based on fast matrix multiplication, which take approximately $O(n^{2.376})$ time. The fastest algorithm for solving general sparse positive semi-definite linear systems is the Conjugate Gradient. Used as a direct solver, it runs in time $O(mn)$ (see Trefethen and Bau 1997, Theorem 28.3).

When the linear system is symmetric and sparse, it is standard to represent the non-zero structure of a matrix A by an unweighted graph G_A that has an edge between vertices $i \neq j$ if and only if $A_{i,j}$ is non-zero. If this graph has a special structure, there may be elimination orderings that accelerate direct solvers. For example, if A is tri-diagonal, in which case G_A is a path graph, then a linear system in A can be solved in time $O(n)$. Similarly, when G_A is a tree, a linear system in A can be solved in time $O(n)$. If the graph of non-zero entries G_A is planar, one can use Generalized Nested Dissection (George 1973; Lipton et al. 1979; Gilbert and Tarjan 1987) to find an elimination ordering under which Cholesky factorization can be performed in time $O(n^{1.5})$ and to produce factors with at most $O(n \log n)$ non-zero entries.

For linear equations that arise when solving elliptic partial differential equations, other techniques supply fast algorithms. For example, Multigrid methods could be effective when applied to some of these linear systems (Briggs et al. 2001), and Hierarchical Matrices run in nearly-linear time when the discretization is well-shaped (Bebendorf and Hackbusch 2003).

However, before the work of the nearly-linear-time Laplacian primitive, no linear solver with complexity better than $O(m^{1.5})$ is known for arbitrary sparse linear systems. So, the Laplacian primitive could open a new page for algorithm design including for fundamental problems such as matching, s-t flows, multicommodity flows, and linear programming. The key step in our attempt to improve the algorithms for these problems is to encode them cleverly by the Laplacian primitive.

Although the Laplacian solver and Laplacian Paradigm are theoretical developments, we are hopeful that they will have considerable practical impact. In fact, each of our algorithms in the suite of the Laplacian Paradigm has been improved since we developed them.

- Using the star-decomposition developed in Elkin et al. (2008), Abraham et al. (2008) further improved the average stretch to a quantity smaller than

$$O(\log n O(\log\log n(\log\log\log n)^3)).$$

- The parameters of local clustering algorithm have subsequently been improved by Andersen et al. (2006) and Andersen and Peres (2009). The former uses personalized Rage-Rank and the latter uses evolving sets to guide the local clustering processing.
- In the original construction of spectral sparsifiers, the $O(1)$ in the exponent of $\log^{O(1)}(n/p)$ is quite large (13 for the running time and 29 for the number of edges). Spielman and Srivastava (2008) reduced the 29 in the exponent of the number of edges in the spectral sparsifier to 1. The running time of their algorithm, using the Laplacian primitive for computing effective resistances, is nearly linear. Recently, Batson et al. (2008) gave a beautiful construction to produce a linearly-sized spectral sparsifier. However, even with polynomial-time complexity, the running time of their algorithm is still far away from being linear.
- In the most exciting advances, Koutis et al. recently improved the running time of the Laplacian solver to $\tilde{O}(m\log n\log(1/\varepsilon))$, where $\tilde{O}$ only hides the ratio of the average stretch of the Abraham-Bartal-Neiman spanning tree to $O(\log n)$, which is $O((\log\log n)^2)$. The recent progress has also greatly enhanced our hope to develop a practical Laplacian solver.

Not Just Numerical Analysis

I would like to conclude by remarking that numerical thinking is not just numerical analysis. It is about the connection between numerical analysis and other fields in computing, as well as about drawing on the fundamental principles & concepts in numerical analysis, and applying them to discover better models for phenomena in computing, and to find faster solutions to problems that have been challenges to us.

Numerical thinking is a creative process of discovering useful connections that may not be apparent. In the 1970s, mathematicians such as Fiedler (1973) and Donath and Hoffman (1972), were able to make a connection between graphs and matrices, which set the stage for spectral graph theory, a field that has made significant strides over the representational connection between graphs and matrices.

Our field has benefitted greatly from the connection between graph properties (such as conductance, mixing time, and connectivity) and algebraic properties (such as the spectral bounds of matrices). Today, computer scientists of our generation have made even broader connections between numerical concepts and network & information concepts (Kleinberg 1999; Brin and Page 1998), between numerical representations and digital representations (Debevec et al. 2000; Daubechies 1992), and between numerical methods and methods for machine learning & data analysis (Deerwester et al. 1990; Donoho 2006).

In our examples, the Laplacian paradigm has not only used numerical concepts such as preconditioning to model graph approximation & graph sparsification, but also used the advancements in graph algorithms to build a new solver for linear systems. Similarly, smoothed analysis has not only extended the studies of stability from numerical analysis to algorithm analysis, but also inspired renewed studies and understanding of the condition number of perturbed matrices (Sankar et al. 2005; Vu and Tao 2007; Rudelson and Vershynin 2006).

Perhaps, the most valuable understanding I have gained from numerical thinking is the interdisciplinary view of the world of computing, as well as the view of our responsibility to both theory and practice. The process of numerical thinking continues to transform my research. I hope this article will encourage more researchers to apply it in their work.

References

I. Abraham, Y. Bartal, O. Neiman. Nearly Tight Low Stretch Spanning Trees. *FOCS* 2008, Pages 781-790.

N. Alon, R. M. Karp, D. Peleg, and D. West. A graph-theoretic game and its application to the k-server problem. *SIAM Journal on Computing*, 24(1):78–100, February 1995.

C. J. Alpert and S.-Z. Yao. Spectral partitioning: the more eigenvectors, the better. In *DAC '95: Proceedings of the 32nd ACM/IEEE conference on Design automation*, pages 195–200. ACM, 1995.

R. Andersen, C. Borgs, J. T. Chayes, J. E. Hopcroft, K. Jain, and V. S. Mirrokni, and S.-H. Teng. Robust PageRank and locally computable spam detection features. In *Fourth International Workshop on Adversarial Information Retrieval on the Web, ACM International Conference Proceeding Series*, 2008 69–76.

R. Andersen, C. Borgs, J. T. Chayes, J. E. Hopcraft, V. S. Mirrokni, and S.-H. Teng. Local computation of pagerank contributions. In Anthony Bonato and Fan R. K. Chung, editors, *WAW*, volume 4863 of *Lecture Notes in Computer Science*, pages 150–165. Springer, 2007.

R. Andersen, F. Chung, and K. Lang. Local graph partitioning using pagerank vectors. *Proceedings: 47th Annual Symposium on Foundations of Computer Science*, pages 475–486, 2006.

R. Andersen, Y. Peres. Finding sparse cuts locally using evolving sets. *STOC*, 2009: 235–244.

S. Arora. Polynomial time approximation schemes for euclidean traveling salesman and other geometric problems. *J. ACM*, 45(5):753–782, 1998.

S. Arora, E. Hazan, and S. Kale. 0(sqrt (log n)) approximation to Sparsest Cut in õ(n^2) time. In *IEEE FOCS' 04*, pages 238–247, 2004.

S. Arora and S. Kale. A combinatorial, primal-dual approach to semidefinite programs. In *ACM STOC '07*, pages 227–236, 2007.

S. Arora, S. Rao, and U. Vazirani. Expander flows, geometric embeddings and graph partitioning. In *ACM STOC '04*, pages 222–231, 2004.

D. Arthur and S. Vassilvitskii. Worst-case and smoothed analysis of the icp algorithm, with an application to the k-means method. In *IEEE FOCS '06*, pages 153–164, 2006.

D. Arthur and B. Manthey and H. Röglin. k-Means Has Polynomial Smoothed Complexity. In *IEEE FOCS*, pages 405–414, 2009.

Nina Balcan and Avrim Blum and Anupam Gupta Approximate clustering without the approximation. *SIAM/ACM SODA* 2009: 1068–1077.

J. Batson, D. A. Spielman, and N. Srivastava. Twice-Ramanujan sparsifiers. Available at http://arxiv.org/abs/0808.0163, 2008.

M. Bebendorf and W. Hackbusch. Existence of H-matrix approximants to the inverse FE-matrix of elliptic operators with L^∞-coefficients. *Numerische Mathematik*, 95(1):1–28, July 2003.

A. A. Benczúr and D. R. Karger. Approximating s-t minimum cuts in O(n^2) time. In *ACM STOC '96*,

M. Bern, J.Gilbert, B. Hendrickson, N. Nguyen, and S. Toledo. Support-graph preconditioners. In *SIAM. J. Matrix Anal. and Appl.* 27(4), 930–951, 2006.

Avrim Blum and John Dunagan. Smoothed analysis of the perceptron algorithm for linear programming. In *SIAM/ACM SODA '02*, pages 905–914, 2002.

E. G. Boman and B. Hendrickson. Support theory for preconditioning. *SIAM Journal on Matrix Analysis and Applications*, 25(3):694–717, 2003.

E. G. Boman, B. Hendrickson, S. Vavasis. Solving epplitic finite element systems in nearly-linear time with support preconditioners.

W. L. Briggs, V. E. Henson, and S. F. McCormick. *A Multigrid Tutorial, 2nd Edition*. SIAM, 2001.

S. Brin and L. Page, The anatomy of a large-scale hypertextual Web search engine, *Proceedings of the seventh international conference on World Wide Web 7*, 107–117, 1998.

J. Cheeger. A lower bound for smallest eigenvalue of laplacian. In *Problems in Analysis*, pages 195–199, In R.C. Gunning editor,, Princeton University Press, 1970.

X. Chen, X. Deng, and S.-H. Teng. Settling the complexity of computing two-player Nash equilibria. In *J. ACM*, 56(3): (2009a)

X. Chen, D. Dai, Y. Du, and S.-H. Teng. Settling the Complexity of Arrow-Debreu Equilibria in Markets with Additively Separable Utilities. In *IEEE FOCS*, 273–282, 2009b.

Siu-Wing Cheng, Tamal K. Dey, Herbert Edelsbrunner, Michael A. Facello, and Shang-Hua Teng. Sliver exudation. *Journal of ACM*, 47:883–904, 2000.

P Chew. There is a planar graph almost as good as the complete graph. In *SCG '86: Proceedings of the second annual symposium on Computational geometry*, pages 169–177. ACM, 1986.

P. Christiano, J. A. Kelner, A. Madry, D. A. Spielman, and S.-H. Teng. Electrical Flows, Laplacian Systems, and Faster Approximation of Maximum Flow in Undirected Graphs http://arxiv.org/abs/1010.2921.

T. Cormen. C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*, 3rd edition.

S. I. Daitch and D. A. Spielman. Support-graph preconditioners for 2-dimensional trusses.

S. I. Daitch and D. A. Spielman. Faster approximate lossy generalized flow via interior point algorithms. In *ACM STOC'08*, pages 451–460, 2008.

Valentina Damerow, Friedhelm Meyer auf der Heide, Harald Räcke, Christian Scheideler, and Christian Sohler. Smoothed motion complexity. In *Proc. 11th Annual European Symposium on Algorithms (ESA'03)*, pages 161–171, 2003.

I. Daubechies, *Ten Lectures on Wavelets*, Society for Industrial and Applied Mathematics, 1992

P. Debevec, T. Hawkins, C. Tchou, H.-P. Duiker, W. Sarokin, and M. Sagar Acquiring the Reflectance Field of a Human Face *SIGGRAPH Conference Proceedings*, 2000

S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by latent semantic analysis *Journal of the American Society for Information Science*, 41 (6), 391407, 1990.

James Demmel. *Applied Numerical Linear Algebra*. SIAM, 1997.

Jian Ding, James R. Lee, and Yuval Peres. Cover times, blanket times, and majorizing measures. *ACM STOC' 11*, 2011.

W. E. Donath and A. J. Hoffman. Algorithms for partitioning of graphs and computer logic based on eigenvectors of connection matrices. *IBM Technical Disclosure Bulletin*, 15:938–944, 1972.

D. L. Donoho, Compressed Sensing, *IEEE Transactions on Information Theory*, 52(4), 12891306, 2006.

S. Dughmi and T. Roughgarden Black-Box Randomized Reductions in Algorithmic Mechanism Design. *IEEE FOCS* 2010: 775–784

Herbert Edelsbrunner, Xiang-Yang Li, Gary Miller, Andreas Stathopoulos, Dafna Talmor, Shang-Hua Teng, Alper Üngör, and Noel Walkington. Smoothing and cleaning up slivers. In *STOC '00*, pages 273–277, 2000.

M. Elkin, Y. Emek, D. A. Spielman, and S.-H. Teng. Lower-stretch spanning trees. *SIAM Journal on Computing*, 32(2):608–628, 2008.

Matthias Englert, Heiko Röglin, and Berthold Vöcking. Worst case and probabilistic analysis of the 2-opt algorithm for the tsp: extended abstract. In *SIAM/ACM SODA '07*, pages 1295–1304, 2007.

S. Even and R. E. Tarjan. Network flow and testing graph connectivity. *SIAM Journal on Computing*, 4(4):507–518, Dec. 1975.

M. Fiedler. Algebraic connectibity of graphs. *Czechoslovak Mathematical Journal*, 23 (98):298–305, 1973.

A. M. Frieze, G. L. Miller, and S.-H. Teng. Separator Based Parallel Divide and Conquer in Computational Geometry. *ACM SPAA* 1992: 420-429

J. A. George. Nested dissection of a regular finite element mesh. *SIAM J. Numer. Anal.*, 10:345–363, 1973.

J. R. Gilbert and R. E. Tarjan. The analysis of a nested dissection algorithm. *Numerische Mathematik*, 50(4):377–404, February 1987.

A. V. Goldberg and S. Rao. Beyond the ow decomposition barrier. J. ACM, 45(5):783797, 1998.

G. H. Golub and M. Overton. The convergence of inexact Chebychev and Richardson iterative methods for solving linear systems. *Numerische Mathematik*, 53:571–594, 1988.

G. H. Golub and C. F. Van Loan. *Matrix Computations*. second edition, 1989.

K. Gremban. *Combinatorial Preconditioners for Sparse, Symmetric, Diagonall y Dominant Linear Systems*. PhD thesis, Carnegie Mellon University, CMU-CS-96-123, 1996.

A. Gulli and A. Signorini. The indexable web is more than 11.5 billion pages. In *WWW '05: Special interest tracks and posters of the 14th international conference on World Wide Web*, pages 902–903. ACM, 2005.

Li-Sha Huang and Shang-Hua Teng. On the approximation and smoothed complexity of leontief market equilibria. In *FAW: Frontiers of Algorithms Workshop*, pages 96–107, 2007.

A. Joshi. *Topics in Optimization and Sparse Linear Systems*, Ph.D. thesis, UIUC, 1997.

A. T. Kalai and A. Samorodnitsky and S.-H. Teng Learning and Smoothed Analysis. In *IEEE FOCS'09*, 395–404, 2009.

J. A. Kelner and E. Nikolova. On the hardness and smoothed complexity of quasi-concave minimization. In *IEEE FOCS'07*, pages 552–563, 2007.

J. A. Kelner and D A. Spielman. A randomized polynomial-time simplex algorithm for linear programming. In *ACM STOC '06*, pages 51–60, 2006.

R. Khandekar, S. Rao, and U. Vazirani. Graph partitioning using single commodity flows. In *ACM STOC '06*, pages 385–390, 2006.

J. Kleinberg. Authoritative sources in a hyperlinked environment. *J. ACM*, 46 (5), 1999, 604–632.

A. Kolla, Y. Makarychev, A. Saberi, and S.-H. Teng Subgraph Sparsification. *ACM STOC'10* 2010.

I. Koutis. G. Miller, and R. Peng. Solving SDD linear systems in time $\tilde{O}(m\log n\log(1/\varepsilon))$. *arXiv:1102.4842*.

I. Koutis. G. Miller, and D. Tolliver. Combinatorial preconditioners and multilevel solvers for problems in computer vision and image prcessing. *International Symp. of Visual Computing*, 1067–1078, 2009.

C.E. Lemke. Bimatrix equilibrium points and mathematical programming. *Management Science*, 11:681–689, 1965.

C.E. Lemke and JR. J.T. Howson. Equilibrium points of bimatrix games. *J. Soc. Indust. Appl. Math.*, 12:413–423, 1964.

X.-Y. Li and S.-H. Teng. Generate sliver free three dimensional meshes. In *ACM-SIAM SODA'01*, pages 28–37, 2001.

R. J. Lipton, D. J. Rose, and R. E. Tarjan. Generalized nested dissection. *SIAM Journal on Numerical Analysis*, 16(2):346–358, April 1979.

T. Leighton and S. Rao. Multicommodity max-flow min-cut theorems and their use in designing approximation algorithms. *Journal of the ACM*, 46(6):787–832, November 1999.

L. Lovasz and M. Simonovits. Random walks in a convex body and an improved volume algorithm. *RSA: Random Structures & Algorithms*, 4:359–412, 1993.

B. Maggs, G. Miller, O. Parekh, R. Ravi, and S. M. Woo. Finding effective support-tree preconditioners. *ACM SPAA* 176–185, 2005.

A. Madry and J. Kelner Faster generation of random spanning trees. *IEEE FOCS'09*, 2009.

Gary L. Miller, Dafna Talmor, Shang-Hua Teng, and Noel Walkington. A delaunay based numerical method for three dimensions: generation, formulation, and partition. In *ACM STOC '95*, pages 683–692, 1995.

G. L. Miller, S.-H. Teng, W. P. Thurston, and S. A. Vavasis. Separators for sphere-packings and nearest neighbor graphs. *J. ACM*, 44(1): 1–29 (1997)

G. L. Miller, S.-H. Teng, W. P. Thurston, and S. A. Vavasis. Geometric Separators for Finite-Element Meshes. *SIAM J. Scientific Computing*, 19(2): 364–386 (1998)

Joseph S. B. Mitchell. Guillotine subdivisions approximate polygonal subdivisions: A simple polynomial-time approximation scheme for geometric tsp, k-mst, and related problems. *SIAM J. Comput.*, 28(4):1298–1309, 1999.

Michael Mitzenmacher and Salil P. Vadhan. Why simple hash functions work: exploiting the entropy in a data stream. *SIAM-ACM SODA* 2008:746–755.

J. Nash. Noncooperative games. *Annals of Mathematics*, 54:289–295, 1951,.

E. Nikolova, J. A. Kelner, M. Brand, and M. Mitzenmacher. Stochastic shortest paths via quasi-convex maximization. In *ESA'06: Proceedings of the 14th conference on Annual European Symposium*, pages 552–563, 2006.

L. Orecchia, L. J. Schulman, U. V. Vazirani, and N. K. Vishnoi. On partitioning graphs via single commodity flows. In *ACM STOC '08*, pages 461–470, 2008.

J. Rief. Efficient approximate solution of sparse linear systems. *Computer and Mathematics with Applications*, 36 (9): 37–58, 1998.

H. Röglin and S.-H. Teng. Smoothed analysis of multiobjective optimization. *IEEE FOCS*, 681–690, 2009.

H. Röglin and B. Vöcking. Smoothed analysis of integer programming. *Math. Program.*, 110 (1):21–56, 2007.

M. Rudelson and R. Vershynin. The littlewood-offord problem and invertibility of random matrices. UC Davis, 2006.

Arvind Sankar, Daniel A. Spielman, and Shang-Hua Teng. Smoothed analysis of the condition numbers and growth factors of matrices. *SIAM Journal on Matrix Analysis and Applications*, to appear, 2005.

Guido Schäfer, Luca Becchetti, Stefano Leonardi, Alberto Marchetti-Spaccamela, and Tjark Vredeveld. Average case and smoothed competitive analysis of the multi-level feedback algorithm. In *IEEE FOCS '03*, page 462, 2003.

A. Sharma, X. Liu, P. Miller, A. Nakano, R. K. Kalia, P. Vashishta, W. Zhao, T. J. Campbell, and A. Haas. Immersive and interactive exploration of billion-atom systems. In *VR '02: Proceedings of the IEEE Virtual Reality Conference 2002*, page 217. IEEE Computer Society, 2002.

J. R. Shewchuk. Tetrahedral mesh generation by Delaunay refinement. In *Proc. 14th Annual ACM Symposium on Computational Geometry*, pages 86–95, 1998.

G. Shklarski and S. Toledo. Rigidity in finite-element matrices: Sufficient conditions for the rigidity of structures and substructures. *SIAM J. Matrix Anal. and Appl.* 30(1): 7–40, 2008.

D. Spielman. Graphs and networks: Random walks and spectral graph drawing. Computer Science, Yale, http://www.cs.yale.edu/homes/spielman/462/lect4-07.pdf, Sept. 18, 2007.

D. A. Spielman and N. Srivastava. Graph sparsification by effective resistances. In *ACM STOC*, pages 563–568, 2008.

D. Spielman and S.-H. Teng. Spectral partitioning works: planar graphs and finite element meshes. In *IEEE FOCS '96*, pages 96–105, 1996.

Daniel A. Spielman and Shang-Hua Teng. Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time. *J. ACM*, 51(3):385–463, 2004.

Daniel A. Spielman and Shang-Hua Teng. Smoothed Analysis: An attempt to explain the behavior of algorithms in practice. *CACM*, 52(10):77–84, 2009.

D. A. Spielman and S.-H. Teng. Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems. *ACM STOC*, 81–90, 2003.

D. A. Spielman and S.-H. Teng. A local clustering algorithm for massive graphs and its application to nearly-linear time graph partitioning. *CoRR*, abs/0809.3232, 2008. Available at http://arxiv.org/abs/0809.3232.

D. A. Spielman and S.-H. Teng. Nearly-linear time algorithms for preconditioning and solving symmetric, diagonally dominant linear systems. *CoRR*, abs/cs/0607105, 2008. Available at http://www.arxiv.org/abs/cs.NA/0607105.

D. A. Spielman and S.-H. Teng. Spectral sparsification of graphs. *CoRR*, abs/0808.4134, 2008. Available at http://arxiv.org/abs/0808.4134.

G. Strang. *Linear Algebra and its Application, 2nd. Ed.* Academic Press, New York, San Francisco, London, 1980.

D. A. Tolliver. *Spectral rounding and image segmentation*. PhD thesis, Pittsburgh, PA, USA, 2006. Adviser-Miller,, Gary L. and Adviser-Collins,, Robert T.

D. A. Tolliver and G. L. Miller. Graph partitioning by spectral rounding: Applications in image segmentation and clustering. In *CVPR '06: Proceedings of the 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 1053–1060. IEEE Computer Society, 2006.

L. N. Trefethen and D. Bau. *Numerical Linear Algebra*. SIAM, Philadelphia, PA, 1997.

P. Vaidya. Solving linear equations with symmetric diagonally dominant matrices by constructing good preconditioners. An invited at IMA, U. Minnesota, Oct. 1991.

R. Vershynin. Beyond hirsch conjecture: Walks on random polytopes and smoothed complexity of the simplex method. In *IEEE FOCS '06*, pages 133–142, 2006.

Van H. Vu and Terence Tao. The condition number of a randomly perturbed matrix. In *ACM STOC '07*, pages 248–255, 2007.

J. H. Wilkinson. Error analysis of direct methods of matrix inversion. *J. Assoc. Comput. Mach.*, 8:261–330, 1961.

D. Zhou, J. Huang and B. Schlkopf. Learning from Labeled and Unlabeled Data on a Directed Graph. *the 22nd International Conference on Machine Learning*, 1041–1048. 2005.

Chapter 16
Fuzzy Logic in Computer Science

Radim Belohlavek, Rudolf Kruse, and Christian Moewes

What Is Fuzzy Logic?

Motivation

To understand fuzzy logic, it is essential to recall the basic motivation that led to its emergence. This motivation, articulated in various forms in the early papers on fuzzy logic by Zadeh (1965, 1973), can briefly be described as follows. Classical logic is appropriate for a formalization of reasoning that involves bivalent propositions such as "5 is a prime number", "age of Jan is 9", or "if x is a positive integer and $y = x + 1$ then y is a positive integer", i.e., propositions which may in principle be true or false. In a similar way, classical sets are appropriate for representing collections (of objects) that have sharp, clear-cut boundaries, such as "the collection of all prime numbers less than 100" or "the collection of all U.S. Senators as of September 1, 2010". For any such collection, an arbitrary given object either is or is not a member of it.

Most propositions which people use to communicate information about the outer world are not bivalent. Such propositions are true to a certain degree, rather than being true or false only. As an example, "it is hot outside" is a proposition whose truth depends on the outside temperature. According to our intuition, the higher the temperature, the truer the proposition. To require that the proposition be bivalent

R. Belohlavek (✉)
Palacky University, Olomouc, Czech Republic
e-mail: radim.belohlavek@acm.org

R. Kruse • C. Moewes
Otto-von-Guericke University, Magdeburg, Germany
e-mail: kruse@iws.cs.uni-magdeburg.de; cmoewes@ovgu.de

E.K. Blum and A.V. Aho (eds.), *Computer Science: The Hardware, Software and Heart of It*, DOI 10.1007/978-1-4614-1168-0_16,

means to require the existence of a particular value, t, such that the proposition is true if the actual temperature is larger than or equal to t and false if the actual temperature is smaller than t. This means that if the actual temperature is, say, $t - 0.01$, we consider the proposition false, while if it is $t + 0.01$, we consider the proposition true. Therefore, if the proposition "it is hot" is regarded as bivalent, an arbitrarily small change in the outside temperature can change its truth value from false to true and vice versa. Needless to say, this contradicts our intuition and the way we use propositions such as "it is hot outside".

Likewise, most collections of objects to which people refer when communicating information do not have sharp, clear-cut boundaries. The membership of objects in such collections is a matter of degree, rather than being a member or not being a member only. The point is well illustrated by a quote from Zadeh's seminal paper (Zadeh 1965):

> More often than not, the classes of objects encountered in the real physical world do not have precisely defined criteria of membership. For example, the class of animals clearly includes dogs, horses, birds, etc. as its members, and clearly excludes objects as rocks, fluids, plants, etc. However, such objects as starfish, bacteria, etc. have an ambiguous status with respect to the class of animals. The same kind of ambiguity arises in the case of a number such as 10 in relation to the "class" of all real numbers which are much greater than 1.
>
> Clearly, the "class of all real numbers that are much greater than 1," or "the class of beautiful women," or "the class of tall men" do not constitute classes or sets in the usual mathematical sense of these terms. Yet, the fact remains that such imprecisely defined "classes" play an important role in human thinking . . .
>
> The purpose of this note is to explore in a preliminary way some of the basic properties and implications of a concept which may be of use in dealing with "classes" of the type cited above. The concept in question is that of a fuzzy set, that is a "class" with a continuum of grades of membership.

Since most propositions about the outer world are not bivalent, classical logic is inadequate to formalize reasoning that involves such propositions. Likewise, since most collections referred to in human communication do not have sharp boundaries, classical sets are inadequate to represent such collections. The main aim of fuzzy logic is to overcome the above-described inadequacies of classical logic and classical sets.

Graded Approach

The principal idea employed by fuzzy logic is to allow for a partially ordered scale of truth values, called also *truth degrees*, which contains the values representing false and true but possibly also other, intermediary truth degrees. That is, the two-element set $\{0, 1\}$ of truth values of classical logic, where 0 and 1 represent false and true, respectively, is replaced in fuzzy logic by a partially ordered scale of truth degrees with the smallest degree being 0 and the largest one being 1. This is known as the *graded approach*. An important example of such scale is the interval $[0, 1]$ of real numbers. A degree from a given scale (e.g., the number 0.9 from $[0, 1]$) that is assigned to a proposition is interpreted as the degree to which the proposition is considered true. For the proposition “it is hot outside”, the higher the outside temperature, the higher the truth degree assigned to this proposition. If 0.9 is assigned to this proposition, it indicates that we consider it being almost hot outside but not completely hot. On the other hand, assigning 0.3 to the same proposition indicates that we consider it being somewhat warm outside but not much. In a similar spirit, scales of truth degrees are used in fuzzy sets to represent degrees to which a given object is a member of a collection with non-sharp boundary. For example, if 0.8 and 0.9 represent degrees to which John and Paul are members of the collection of tall men, respectively, it indicates that both are considered almost tall and that Paul is a little bit taller than John.

Controversies

It is clear from the discussion above that fuzzy logic departs from two important traditions of science – the principle of bivalence and the principle that all scientifically relevant concepts are precise and clear-cut. This departure brought up several fundamental issues at stake, which have been, and continue to be, an object of controversy. Two such issues are briefly described in this section. Another one is discussed in section “Fuzzy Logic and Probability”.

The basic idea of fuzzy logic, i.e., that propositions may have intermediary truth degrees, represents a radical departure from one of the basic principles of classical logic and exact sciences – the *principle of bivalence*, according to which every proposition is either true or false. Various ramifications of admitting intermediary truth degrees have been examined in a number of papers, see Smith (2009) for numerous references. Some of the papers pose interesting problems and challenges for fuzzy logic. Quite often, however, the authors of the critical papers are not familiar enough with the principles of fuzzy logic and their analyzes are based on various types of misunderstanding and misconception. Among the critiques of fuzzy logic is a number of attempts to prove that fuzzy logic leads to counterintuitive results and even to contradictions. The best known such critique are Elkan’s papers (Elkan 1993, 1994), the second of which appeared in a special issue of *IEEE Expert* along with responses to it. The central claim of Elkan’s critique was that “as a formal system, a standard version of fuzzy logic collapses mathematically to two-valued logic.” This claim is the content of two theorems presented in Elkan (1993, 1994). In both cases, proofs of the theorems are quite long. Since it is common to take the length of a proof as a measure of profundity of the proven theorem, Elkan’s theorems may look on the surface as quite profound. However, a close examination of the theorems demonstrates the contrary. Namely, Belohlavek and Klir (2007) present short proofs of both theorems and by using these proofs they show that axioms upon which Elkan’s theorems are based define formal systems that are strange to fuzzy logic and are not capable of dealing with fuzziness.

The second controversy relates to a long-standing tradition in science according to which *all scientifically relevant concepts are precise and clear-cut*. Contrary to this tradition, fuzzy logic claims to provide us with a mathematical tools to model and process concepts that are not clear-cut. Namely, fuzzy logic uses scales of truth values to capture the meaning of propositions and collections which involve non-clear-cut concepts such as "hot", "tall", and the like. To capture the meaning of such terms, referred to as *vague terms*, in an appropriate way is quite an intricate issue. This brings up an important question whether the approach of fuzzy logic, based on scales of truth degrees, is appropriate. Such question is very complex and has many facets, ranging from philosophy and mathematics to psychology and cognitive science. Thus far, this question has not been decisively answered and is currently a subject of discussion (van Deemter 2010; Smith 2009). Nevertheless, the use of fuzzy logic is supported by at least the following three arguments. First, fuzzy logic is rooted in the intuitively appealing idea that the truth of propositions used by humans is a matter of degree. An important consequence is that the basic principles and concepts of fuzzy logic are easily understood. Second, fuzzy logic has led to many successful applications, including many commercial products, in which the crucial part relies on representing and dealing with statements in natural language that involve vague terms. Third, fuzzy logic is a proper generalization of classical logic and, follows an agenda similar to that of classical logic, and has already been highly developed. An important consequence is that fuzzy logic extends the rich realm of applications of classical logic by applications in which the bivalent character of classical logic is a limiting factor.

Fuzzy Logic and Probability

Ever since the publication of (Zadeh 1965), the relationship between fuzzy logic and probability theory has been an object of another controversy. The various facets of this relationship have been discussed in many papers, including those contained in the special issues of *Computational Intelligence* (Vol. 4, No. 2, 1988), *IEEE Transactions on Fuzzy Systems* (Vol. 2, No. 1, 1994), and *Technometrics* (Vol. 37, No. 3, 1995). An extensive discussion on this topic comes as no surprise because both fuzzy logic and probability address the phenomenon of uncertainty and both use the real unit interval [0, 1]. The central questions of the debate include:

How does fuzzy logic relate to probability theory?
Is uncertainty the same as randomness?
Does the notion of probability exhaust all our notions of uncertainty?

The earliest paper discussing the relationship between fuzzy logic and probability is (Loginov 1966) in which the author suggests that membership degrees of fuzzy sets may be interpreted as conditional probabilities. This or a similar view has later been adopted by many people. Several leading researchers, including Cheeseman (1988a,b) and Lindley (1987), were repeatedly criticizing fuzzy logic on the ground that probability methods alone, and Bayesian methods in particular, are sufficient for representation and management of any type of uncertainty. As an illustration, the following is a quote from (Lindley 1987):

> The only satisfactory description of uncertainty is probability. By this I mean that every uncertainty statement must be in the form of a probability; that several uncertainties must be combined using the rules of probability; and that the calculus of probabilities is adequate to handle all situations involving uncertainty. … We speak of "the inevitability of probability."

In Sect. 16, Lindley concludes:

> … probability is the only sensible description of uncertainty and is adequate for all problems involving uncertainty. All other methods are inadequate. … My challenge that anything that can be done with fuzzy logic, …, or any other alternative to probability, can better be done with probability, remains.

On the other hand, it has been pointed out many times, see e.g., (Klir 1989) and (Kosko 1990), that fuzzy logic studies a type of uncertainty that is fundamentally different from that studied by probability theory. As an example, take the proposition "Peter is a tall man." As explained above, fuzzy logicians consider this as a many-valued (fuzzy) proposition, i.e., a proposition whose truth degree may be any degree from [0, 1] (or from another appropriate scale of truth degrees). The higher the degree, the truer the proposition. The graded nature of such propositions reflects the graded nature of human concepts such as the concept of a tall man. Note that the graded nature of human concepts was confirmed by many experiments in the psychology of concepts (Belohlavek and Klir 2011). Considering the proposition "Peter is a tall man." as a bivalent proposition (yes-or-no proposition) is inadequate. For example, the question "Is the proposition true, but answer 'yes' or 'no' only?", is inappropriate because it distorts the meaning of the concept of a tall man, namely it distorts its fuzziness. When probability theorists suggest that truth degrees of propositions are (conditional) probabilities, they assume that the propositions themselves are bivalent and that the truth degree measures a person's (subjective) uncertainty of whether the proposition is true, i.e., whether the truth degree of the proposition is 1. Clearly, this view is very different from the view of fuzzy logicians. Because fuzzy propositions are considered bivalent in this view, the view is considered fundamentally inadequate by fuzzy logicians.

The above considerations point to the fact that fuzzy logic and probability study different types of uncertainty, that these types are complementary and are both important in human action. Hence, fuzzy logic and probability theory should be looked at as complementary rather than competitive theories. This situation was recognized in an early paper by Zadeh (1968). In order to extend the applicability of probability theory to account for fuzzy events such as "high inflation rate", Zadeh proposed to generalize the concept of a probability space by allowing events to be fuzzy sets rather than ordinary sets of elementary events. The need for extensions of probability theory that take into account fuzziness of natural language expressions, which is particularly emphasized by the demand for natural language interfaces in web search, has recently been pointed out in several papers by Zadeh (2002, 2006).

In (Zadeh 2002) the following examples of simple problems are presented for which probability theory does not provide solutions:

Most Swedes are tall. Most Swedes are blond. What is the probability that a Swede picked at random is tall and blond?

Usually Robert returns from work at about 6 p.m. What is the probability that he is home at 6:30 p.m.?

A box contains about 20 balls of various sizes. A few are small and several are large. What is the probability that a ball drawn at random is neither large nor small?

In view of these examples, it becomes apparent that to base probability theory on bivalent logic results in a fundamental limitation and that, naturally, probability theory should be based on fuzzy logic. Such a conclusion presents a serious challenge for research in the foundations of probability theory.

Various Meanings of "Fuzzy Logic"

The term "fuzzy logic", coined by Goguen (1968), is used in several meanings. In its common-sense meaning, the term refers to formal and informal principles and methods of reasoning that involve vaguely defined concepts (concepts without clear-cut boundaries) that are based on the graded approach.

Two other meanings are frequently used, fuzzy logic in the narrow sense and fuzzy logic in the broad sense. Fuzzy logic in the narrow sense, called also mathematical fuzzy logic (Hájek 2006), develops deductive systems of logic very much in the style of classical mathematical logic. When the term fuzzy logic is used in the broad sense, it refers to an attempt to emulate human reasoning in natural language and includes aspects that are beyond the usual scope of mathematical logic. Fuzzy logic in the narrow and broader sense are discussed in more detail in section "Fuzzy Logic as Logic".

Basic Concepts of Fuzzy Logic

Truth Degrees and Truth Functions of Logical Connectives

As mentioned above, fuzzy logic uses a scale, denoted here by L, of truth degrees. A common choice for L is $[0, 1]$ (real unit interval) and unless stated otherwise, we assume $L = [0, 1]$ throughout this section. In general, L is usually assumed to be a complete lattice bounded by 0 and 1. As in classical logic (where $L = \{\ 0, 1\}$), the scale needs to be equipped with (truth functions of) logical connectives such as conjunction, implication, etc. Unlike classical logic, where there truth functions are simply derived from the use of connectives in language and are unique (form example, "φ and ψ" is true if and only if both φ and ψ are true), fuzzy logic does not have unique truth functions of logical connectives. Namely, if there is no obvious way to define the truth degree of proposition "φ and ψ" given that the truth degree of φ and ψ are 0.7 and 0.8, respectively. Therefore, rather than defining a particular truth function of conjunction ("the right function"), fuzzy logic accepts as appropriate any truth function which satisfies certain conditions that come from intuitive requirements as well as from particular application contexts. For example, a truth function $\otimes$ of conjunction is a binary function $\otimes : L \times L \rightarrow L$ which needs to satisfy at least the following conditions:

$$\begin{aligned}
&a_1 \leq a_2 \text{ and } b_1 \leq b_2 \text{ implies } a_1 \otimes b_1 \leq a_2 \otimes b_2, && \text{(monotonicity)}\\
&a \otimes b = b \otimes a, && \text{(commutativity)}\\
&a \otimes (b \otimes c) = (a \otimes b) \otimes c, && \text{(associativity)}\\
&a \otimes 1 = 1 \otimes a = a, a \otimes 0 = 0 \otimes a = 0, && \text{(boundary conditions)}
\end{aligned}$$

which are certainly intuitively appealing properties of conjunction. A function $\otimes$ on $L = [0, 1]$ satisfying these conditions is called a *t*-norm (Klement et al. 2000). The *t*-norms used in fuzzy logic are usually continuous (or at least left-continuous). The basic continuous *t*-norms are Gödel (maximum), Goguen (product), and Łukasiewicz *t*-norm, which are defined as follows:

$$\text{Gödel: } a \otimes b = \min(a, b), \tag{16.1}$$

$$\text{Goguen: } a \otimes b = a \cdot b, \tag{16.2}$$

$$\text{Łukasiewicz: } a \otimes b = \max(a + b - 1, 0). \tag{16.3}$$

Namely, any continuous *t*-norm can be obtained from the basic ones by so-called ordinal sum (Hájek 1998; Klement et al. 2000). *t*-norms have been extensively studied in the literature and various classes of *t*-norms, including classes of parameterized *t*-norms such as $a \otimes_\lambda b = 1 - \min\{1, [(1-a)_\lambda + (1-b)_\lambda]_\lambda\}$ for $\lambda \in [0, \infty)$ are described, e.g., in Gottwald (2001), Klement et al. (2000) and Klir and Yuan (1995).

In general, a truth function of an n-ary logical connective is a function $c : L^n \rightarrow L$. As in classical logic, further connectives such as disjunction, implication, or negation, are used in fuzzy logic. Due to limited scope we do not discuss the truth functions of these connectives here and refer the reader e.g., to Gottwald (2001) and Klir and Yuan (1995). An important question of a relationship between the truth functions of logical functions, such as the relationship between conjunction and implication, is discussed in section "Fuzzy Logic as Logic".

In addition to the connectives mentioned so far, fuzzy logic used various other connectives. For illustration, we mention linguistic modifiers and averaging functions. Modifiers are unary functions $m : [0,1] \to [0,1]$ which are thought of as the truth functions of unary connectives, called linguistic hedges Zadeh (1973, 1975), such as "very", "highly", "more or less", "somewhat", etc. Linguistic hedges are employed in linguistic rules such as "If temperature is very high, then …". A simple class of modifiers is given by

$$m_\lambda(a) = a_\lambda$$

for $a \in [0,1]$. For $a \in (0,1)$, the modifier is an increasing function and corresponds to linguistic hedges such as "more or less" or "somewhat". For $a \in (1,\infty)$, the modifier is a decreasing function and corresponds to intensifying linguistic hedges such as "very" or "highly". Averaging functions are defined as n-ary functions $c : [0,1]^n \to [0,1]$ that are non-decreasing, idempotent, and usually continuous and symmetric. Because they satisfy

$$\min(a_1, \ldots, a_n) \leq c(a_1, \ldots, a_n) \leq \max(a_1, \ldots, a_n)$$

and because min and max are "the largest (truth function of) conjunction" and "the least (truth function of) disjunction", averaging functions are thought of as filling a gap between conjunctions and disjunctions. As simple example is the arithmetical average $c(a,b) = \frac{a+b}{2}$. According to common sense, a person's financial wealth depends on whether his assets have good liquidity and his investments are good. Naturally, the degree $W(x)$ to which a person x is financially wealthy is obtained from the degrees $L(x)$ (good liquidity) and $I(x)$ (good investment) by means of an averaging function (e.g., $W(x) = \frac{L(x)+I(x)}{2}$) rather than a conjunction (e.g., $W(x) = \min\{L(x), I(x)\}$) or disjunction (e.g., $W(x) = \max\{L(x), I(x)\}$). Note that neither the modifiers nor the averaging functions have a counterpart in classical logic (modifiers are degenerate in classical logic, the only one is the identity function mapping 0 to 0 and 1 to 1; classical truth degrees cannot be averaged).

Fuzzy Sets and Fuzzy Relations

The concept of a fuzzy set generalizes the concept of a (characteristic function of a) classical set. A fuzzy set A in a universe U is defined as a mapping $A : U \to L$, i.e., A assigns to every element u from U a degree $A(u)$ from a scale L of truth degrees, called the degree of membership of u to A. If $L = [0,1]$, one usually speaks of standard fuzzy sets. Clearly, if $L = \{0,1\}$, we get the notion of a characteristic notion of an ordinary set.

The notions and operations related to fuzzy sets include both the counterparts of those from classical sets as well as new ones. An important example of the latter is the concept of an α-cut, which is defined for $\alpha \in L$ and a fuzzy set A as the ordinary subset ${}^{\alpha}A$ of U defined by ${}^{\alpha}A = \{u \in U \mid A(u) \geq \alpha\}$. A fuzzy set A is uniquely

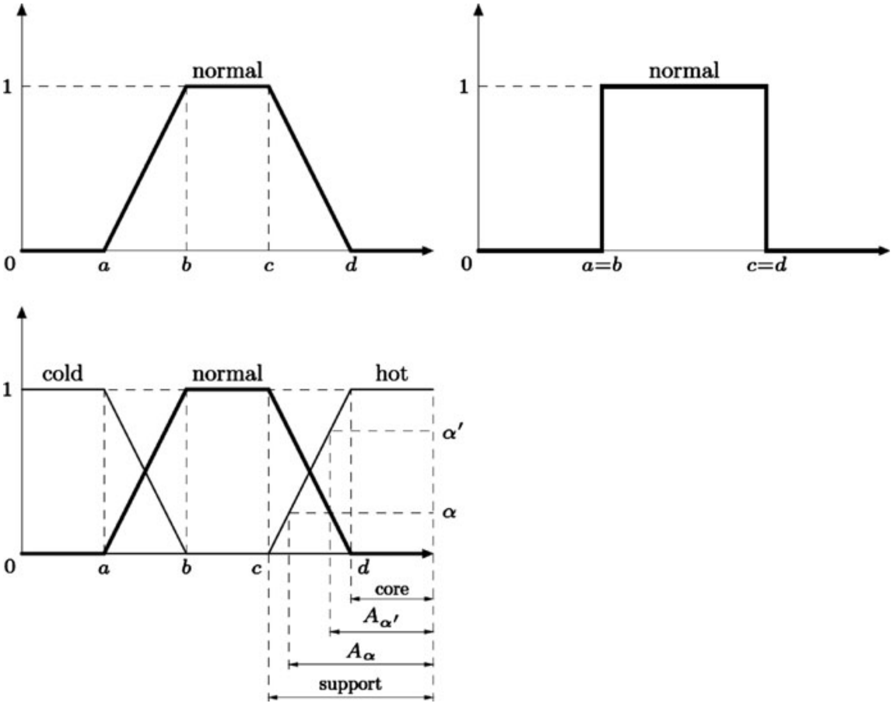


Fig. 16.1 Concept of fuzzy set

represented by the collection $\{{}^{\alpha}A \mid \alpha \in L\}$ of all of its α-cuts and this representation connects fuzzy sets with ordinary sets. The top part of Fig. 16.1 shows a fuzzy set representing the concept "normal" (temperature) versus a classical set representing the same concept. The bottom part shows three fuzzy sets, representing "cold", "normal", and "hot", and illustrates the concepts of an α-cut and support of a fuzzy set defined as $\text{supp}(A) = \{u \in U \mid A(u) > 0\}$.

Every logical n-ary connective c on L induces a corresponding n-ary operation, defined component-wise. For example, if c is the truth function min of Gödel conjunction, the corresponding operation, called the standard intersection of fuzzy sets and denoted by $\cap$, is defined by

$$(A \cap B)(x) = \min(A(x), B(x)).$$

Relations on fuzzy sets can be both ordinary relations, such as the inclusion $\subseteq$ of fuzzy sets defined by $A \subseteq B$ if and only if $A(u) \leq B(u)$ for each $u \in U$. However, one may in general consider fuzzy versions of these relations, such as a degree of inclusion of fuzzy sets, which play an important role in fuzzy set theory.

Fuzzy relations are defined as fuzzy sets in Cartesian products. For example, a binary relation between sets U and V is a mapping $R : U \times V \to L$ with $R(u, v)$ being interpreted as a degree to which u is related to v. Among the several types of fuzzy relations used in applications, fuzzy equivalences (called also similarity relations) are perhaps the most important. A fuzzy relation $E : U \times U \to L$ is called a fuzzy equivalence if the following conditions generalizing the ordinary reflexivity, symmetry, and transitivity hold true:

$$\begin{aligned} E(u,u) &= 1, \\ E(u,v) &= E(v,u), \\ E(u,v) \otimes E(v,w) &\leq E(u,w), \end{aligned}$$

where $\otimes$ is a truth function of conjunction.

Various particular types of fuzzy sets and fuzzy relations are used in applications of fuzzy logic and were studied in the literature. Due to lack of space we omit details and refer the reader to numerous books on fuzzy sets and their applications, e.g., to Belohlavek (2002), Gottwald (2001), Klir and Yuan (1995) and Kruse et al. (1994).

Fuzzy Logic as Logic

Is there any logic in "fuzzy logic", i.e., is it possible to develop a deductive system for reasoning which involves degrees of truth? What are the corresponding concepts of consequence, provability, completeness and what properties do they have? As was mentioned in section "What Is Fuzzy Logic?", these question are addressed by fuzzy logic in the narrow sense. This section provides an introduction to the basic concepts involved.

Fuzzy Logic as Many-Valued Logic

Logics with more than two truth values, so-called many-valued logics, were studied in the field of mathematical logic since 1930s, see e.g., Gottwald (2001). Fuzzy logic can be considered a particular many-valued logic whose agenda is driven by the interpretation of truth values as truth degrees. Fuzzy logic uses many-valued counterparts of logical connectives of classical logic, as was discussed in section "Truth Degrees and Truth Functions of Logical Connectives". In addition, fuzzy logic is truth functional. That is, if $||\varphi||$ and $||\psi||$ denote the truth degrees of formulas φ and ψ, the truth degree $||\varphi \& \psi||$ of the conjunction of φ and ψ is determined by

$$||\varphi \& \psi|| = ||\varphi|| \otimes ||\psi|| \tag{16.4}$$

where $\otimes$ is a truth function of conjunction; and the same for other connectives.

Since in fuzzy logic, there are many possible choices of the truth functions of logical connectives (section "Truth Degrees and Truth Functions of Logical Connectives"), it is important to ask which combinations of truth functions are appropriate. An important argument regarding the choice of the truth functions of conjunction and implication comes from Goguen (1968) who showed that this question is connected to the rule of *modus ponens*. In particular, if one wants to have a good rule of *modus ponens* (yielding as much as possible but still sound), the truth functions $\otimes$ of conjunction and $\to$ of implication need to satisfy

$$a \otimes b \leq c \text{ if and only if } a \leq b \to c, \tag{16.5}$$

called the adjointness condition. For example, if $\otimes$ is a continuous (or even a left-continuous) t-norm, the unique $\to$ satisfying (16.5), called the residuum of $\otimes$, is given by

$$a \to b = \sup\{z \mid a \otimes z \leq b\}.$$

In particular, the residua of Gödel, Goguen, and Łukasiewicz t-norms, see (16.1)–(16.3), are given by

$$\begin{aligned}
\text{Gödel: } a \to b &= \begin{cases} 1 & \text{if } a \leq b, \\ b & \text{otherwise,} \end{cases} \\
\text{Goguen: } \mathrm{a} \to \mathrm{b} &= \begin{cases} 1 & \text{if } a \leq b, \\ \frac{b}{a} & \text{otherwise,} \end{cases} \\
\text{Łukasiewicz: } a \to b &= \min(1 - a + b, 1).
\end{aligned}$$

Ordinary-Style Calculi

Two basic types of fuzzy logical calculi can be distinguished. The first one are called ordinary-style calculi. Except for the fact that they allow more than two truth degrees, their its agenda is practically the same as that of classical logic. For example, formulas are defined as usual (starting from atomic formulas and applying logical connectives), a theory is a set of formulas, a proof from a theory T is a sequence of formulas which are either from T or result by application of a deduction rule to preceding formulas, etc. Due to truth functionality, the truth degree of a formula is defined as usual, cf. (16.4), given that particular structure $\mathbf{L}$ of truth degrees is chosen, i.e., a set L of truth degrees and truth functions of logical connectives from the language of the particular logical calculus. A tautology w.r.t. a class $\mathcal{L}$ of structures of truth degrees if for every structure $\mathbf{L} \in \mathcal{L}$, φ has truth degree 1 for every evaluation using truth degrees and logical connectives from $\mathbf{L}$. To illustrate ordinary-style completeness, consider the completeness theorem of propositional BL-logic Hájek (1998) that was proved in Cignoli et al. (2000). Given the axioms of BL-logic, the following conditions are equivalent for any formula φ:

1. φ is provable.
2. φ is a tautology w.r.t. the class of algebras which consist of [0, 1], a continuous t-norm, and its residuum.
3. φ is a tautology w.r.t. the class of BL-algebras (particular lattices equipped with operations $\otimes$ and $\rightarrow$, the algebraic counterparts of BL-logic).

For more information we refer to Gottwald (2001) and Hájek (1998).

Graded-Style (Pavelka-Style) Calculi

Graded-style calculi were introduced in a seminal paper by Pavelka (1979). Unlike ordinary-style calculi, the graded-style calculi works with formulas to which truth degrees are "attached". A pair $\langle\varphi, a\rangle$ carries a syntactical information that formula φ be true to degree at least a. For example, a theory is a set consisting of such pairs $\langle\varphi, a\rangle$ which specify that φ is assumed to be true to degree at least a. A deductive rule has two components, one working on formulas, the other working on truth degrees. For example, the rule of *modus ponens* applied to $\langle\varphi \Rightarrow \psi, a\rangle$ and $\langle\varphi, b\rangle$ yields a pair $\langle\psi, a \otimes b\rangle$ and reads as follows: If $\varphi \Rightarrow \psi$ and φ are true to degree at least a and b, respectively, ψ is true to degree at least $a \otimes b$. One then introduces the concept of a degree $|\varphi|_T$ to which formula φ is provable from theory T (supremum of as over all $\langle\varphi, a\rangle$ which can be obtained from the axioms and T using deduction rules) and the concept of a degree $\|\varphi\|_T$ to which φ is (semantically) entailed by T (infimum of truth degrees of φ in all models of T). A completeness theorem then says

$$|\varphi|_T = \|\varphi\|_T,$$

i.e., degree of probability equals degree of entailment. For further information including various particular graded-style calculi we refer to Belohlavek and Vychodil (2005, 2006), Gerla (2001) and Hájek (1998).

Fuzzy Logic in a Broad Sense

Note that from a general viewpoint of logic as a discipline studying human reasoning, fuzzy logic in the broad sense also fits the picture of fuzzy logic as logic. As mentioned in section "What Is Fuzzy Logic?", fuzzy logic in the broad sense attempts to emulate human reasoning. Conceptually, fuzzy logic in the broad sense is being developed in numerous papers by Zadeh (1973, 1975, 1979, 2006, 2008). Parts of fuzzy logic in the broad sense are highly developed and have numerous applications, for example the rule-based systems employed in fuzzy control, discussed in sections "Fuzzy Logic and Control" and "Success of Mamdani Control in Automobile Industry".

Note however, that traditional logical aspects of logic are as a rule of little concern in those developments, but see Hájek's chapter on logical analysis of the compositional rule of inference in (Hájek 1998) and also (Novák et al. 1999). From this point of view, fuzzy logic in the broad sense is at an early stage of development.

Fuzzy Logic and Control

The biggest success of fuzzy logic in the field of industrial and commercial applications has been achieved with *fuzzy controllers*. Fuzzy control is a way of defining a nonlinear table-based controller whereas its nonlinear transition function can be defined without specifying every single entry of the table individually. Fuzzy control does not result from classical control engineering approaches. In fact, its roots can be found in the area of rule-based systems. Fuzzy controllers simply comprise a set of vague rules that can be used for knowledge-based interpolation of a vaguely defined function.

Suppose we consider a technical system. For this system, we dictate a desired behavior. Generally a time-dependent *output variable* must reach a desired set value. The output is influenced by a *control variable* which we can manipulate. Finally, there exists a time-dependent *disturbance variable* that influences the output as well. The current control value is usually determined based on the current measurement values of the output variable ξ, the variation of the output $\Delta\xi = \frac{d\xi}{dt}$ and further variables.

Hereafter we will refer to input variables $\xi_1 \in X_1, \ldots, \xi_n \in X_n$ and one control variable $\eta \in Y$. The solution of a control problem is a suitable control function $\varphi : X_1 \times \ldots \times X_n \to Y$ that determines an appropriate control value $y = \varphi(\mathbf{x})$ for every input tuple $\mathbf{x} = (x^{(1)}, x^{(2)}, \ldots, x^{(n)}) \in X_1 \times \ldots \times X_n$. In classical control engineering, φ is commonly determined by solving a set of differential equations. It is very often out of the question to specify an exact set of differential equations. Note that human beings, however, are greatly able to control certain processes without knowing about higher mathematics.

Simulating the behavior of a human "controller" can be done by questioning the individual directly. An alternative would be extract essential information by observing the controlled process. The result of such *knowledge-based analysis* is a set of *linguistic rules* that control the process. Linguistic rules comprise a premise and a conclusion. The former relates to a fuzzy description of the crisp measured input, where the latter defines a suitable fuzzy output. Thus we need to formalize mathematical descriptions of the linguistic expressions used in the rules. Furthermore initialized rules need to be accumulated to result in one fuzzy output value. Finally, a crisp output value must be computed from the fuzzy one. The whole architecture for that knowledge-based model of a fuzzy controller is shown in Fig. 16.2.

The *fuzzification interface* operates on the current input value $\mathbf{x}_0$. If needed, $\mathbf{x}_0$ is mapped into a suitable domain, e.g., normalization to the unit interval. It also

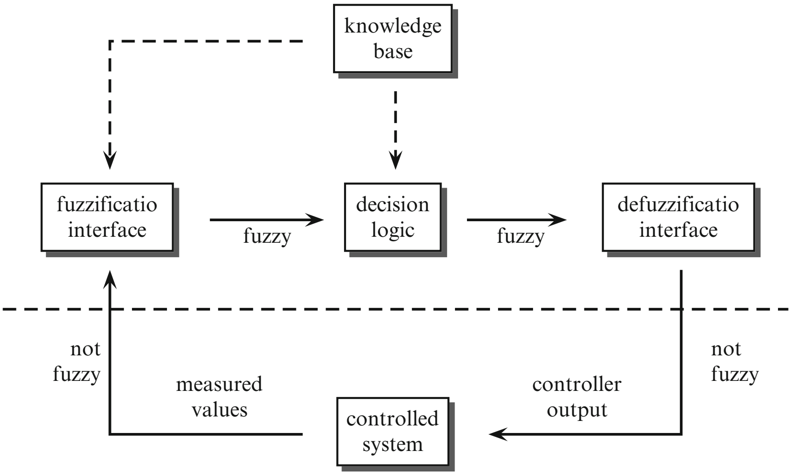


Fig. 16.2 Architecture of a fuzzy controller

transforms $\mathbf{x}_0$ into a linguistic term or fuzzy set. The *knowledge base* comprises the *data base*, i.e., all pieces of information about variable ranges, domain transformations, and the fuzzy sets with their corresponding linguistic terms. Moreover, it also contains a *rule base* storing the linguistic rules for controlling. The *decision logic* determines the output value of the corresponding measured input using the knowledge base. The *defuzzification interface* produces the crisp output value given the fuzzy output.

There exist two fundamentally different approaches to fuzzy control. Both of them are motivated intuitively (see the next two sections). We will see in section "Approximate Reasoning" that a fuzzy controller based on logical implications results in completely different methods of computation.

Mamdani-Assilian Controller

In 1975, the first model of a fuzzy controller was created by Ebrahim "Abe" Mamdani and his student Sedrak Assilian (Mamdani and Assilian 1975). Mamdani and Assilian developed their idea application-driven to control a steam engine based on human expert knowledge.

Here, the knowledge of an expert must be expressed by linguistic rules. First, for the set X_1, p_1 fuzzy sets $\mu_1^{(1)}, \dots, \mu_{p_1}^{(1)} \in \mathcal{F}(X_1)$ must be defined. Accordingly, each fuzzy set is named with a suitable linguistic term. Second, X_1 is partitioned by its fuzzy sets. To be able to interpret each fuzzy set as fuzzy value or fuzzy interval, it is favorable to only use unimodal membership functions. Also, fuzzy sets of one partition should be disjoint, i.e., they satisfy

$$i \neq j \Rightarrow \sup_{x \in X_1} \left\{ \min\left\{ \mu_i^{(1)}(x), \mu_j^{(1)}(x) \right\} \right\} \leq 0.5.$$

Having divided X_1 into p_1 fuzzy sets $\mu_1^{(1)}, \ldots, \mu_{p_1}^{(1)}$, we partition the remaining sets $X_2, \ldots, X_n$ and Y in the same manner. Finally, these fuzzy partitions and the linguistic terms associated with the fuzzy sets correspond to the data base in our knowledge base.

The rule base is specified by rules of the form

$$\textbf{if } \xi_1 \textbf{ is } A^{(1)} \textbf{ and} \ldots \textbf{and } \xi_n \textbf{ is } A^{(n)} \textbf{ then } \eta \textbf{ is } B \tag{16.6}$$

whereas $A^{(1)}, \ldots, A^{(n)}$ and B represent linguistic terms corresponding to fuzzy sets $\mu^{(1)}, \ldots, \mu^{(n)}$ and μ, respectively, according to fuzzy partitions of $X_1 \times \ldots \times X_n$ and Y. Hence the rule base comprises k control rules

$$R_r : \textbf{if } \xi_1 \textbf{ is } A_{i_{1,r}}^{(1)} \textbf{ and} \ldots \textbf{and } \xi_n \textbf{ is } A_{i_{n,r}}^{(n)} \textbf{ then } \eta \textbf{ is } B_{i_r}, \quad r = 1, \ldots, k.$$

Remark that these rules are not regarded as logical implications. They rather define $\eta = \varphi(\xi_1, \ldots, \xi_n)$ piecewise where

$$\eta \approx \begin{cases} B_{i_1} & \text{if } \xi_1 \approx A_{i_{1,1}}^{(1)} \text{ and } \ldots \text{ and } \xi_n \approx A_{i_{n,1}}^{(n)}, \\ \vdots & \vdots \\ B_{i_k} & \text{if } \xi_1 \approx A_{i_{1,k}}^{(1)} \text{ and } \ldots \text{ and } \xi_n \approx A_{i_{n,k}}^{(n)}. \end{cases}$$

Since the rules are treated as *disjunctive*, we can say that the control function φ is obtained by knowledge-based interpolation.

Observing a measurement $\mathbf{x} \in X_1 \times \ldots \times X_n$ the decision logic applies each R_r separately. It computes the degree to which $\mathbf{x}$ fulfills the premise of R_r, i.e., the degree of applicability

$$\alpha_r \overset{\text{def}}{=} \min\left\{ \mu_{i_{1,r}}^{(1)}(x^{(1)}), \ldots, \mu_{i_{n,r}}^{(n)}(x^{(n)}) \right\}. \tag{16.7}$$

"Cutting off" the output fuzzy set μ_{i_r} of rule R_r at α_r leads to the rule's output fuzzy set:

$$\mu_{\mathbf{x}}^{\mathrm{o}(R_r)}(y) = \min\{\alpha_r, \mu_{i_r}(y)\}. \tag{16.8}$$

Having computed all α_r for $r = 1, \ldots, k$, the decision logic combines all $\mu_{\mathbf{x}}^{\mathrm{o}(R_r)}$ applying the t-conorm maximum in order to get the overall output fuzzy set

$$\mu_{\mathbf{x}}^{\mathrm{o}}(y) = \max_{r=1,\ldots,k} \{\min\{\alpha_r, \mu_{i_r}(y)\}\}. \tag{16.9}$$

In control engineering, a crisp control value is needed. Therefore $\mu_{\mathbf{x}}^{\mathrm{o}}$ is forwarded to the defuzzification interface. Here, it depends on the kind of method that is implemented to defuzzify $\mu^{\mathrm{o}}_{\mathbf{x}}$. The most well-known approaches are the max criterion method, the mean of maxima (MOM) method and the center of gravity (COG) method. Using the first approach, simply an arbitrary value $y \in Y$ is chosen for which $\mu^{\mathrm{o}}_{\mathbf{x}}(\,y)$ reaches a maximum membership degree. Picking a random value leads to a nondeterministic control behavior which is usually undesired. The MOM method choses the mean value of the set of elements $y \in Y$ resulting in maximal membership degrees. The defuzzified control value η might not even be in the set which can lead to unexpected control actions. The COG method defines the value located under the center of gravity of the area $\mu^{\mathrm{o}}_{\mathbf{x}}$ as control value η, i.e.,

$$\eta = \left(\int_{y \in Y} \mu^{\mathrm{o}}_{\mathbf{x}}(y) \cdot y \, \mathrm{d}y\right) \Big/ \left(\int_{y \in Y} \mu^{\mathrm{o}}_{\mathbf{x}}(y) \, \mathrm{d}y\right). \tag{16.10}$$

In most control applications, this method shows smooth control behaviors. However, it might even lead to counterintuitive results as well. For a more profound discussion about defuzzification, see e.g., Kruse et al. (1994).

Let us conclude this type of controller by analyzing the form of linguistic rules again. Regarding (16.8), it is clear that the minimum is used as fuzzy implication. Obviously this does not coincide with its crisp counterpart. Just consider $p \rightarrow q$ knowing that p is false. Then $p \rightarrow q$ is true regardless of the truth value of q in classical propositional logic. However, $\min\{0, q\}$ is always 0. One way to justify the heuristic of Mamdani and Assilian is to replace the concept of implication by the one of *association* (Cordón et al. 1999). We say that for a rule R_r an output fuzzy set B_{i_r} is associated with n input fuzzy sets $A^{(j)}_{i_{j,r}}$ for $j = 1, \ldots, n$. This association is modeled by a fuzzy conjunction, e.g., the t-norm min.

We retrieve Mamdani's heuristics by extensionality assumptions (Klawonn et al. 1995; Klawonn and Kruse 1993). If the fuzzy relation R relating the $x^{(j)}$ and y satisfies some extensionality properties, then Mamdani's approach is derived in the same way. Let E and E' be two similarity relations defined on the domains X and Y of x and y, respectively. The extensionality of R on $X \times Y$ thus means

$$\begin{aligned} &\forall x \in X : \forall y, y' \in Y : R(x,y) \otimes E'(y,y') \leq R(x,y'), \\ &\forall x, x' \in X : \forall y \in Y : R(x,y) \otimes E(x,x') \leq R(x',y). \end{aligned} \tag{16.11}$$

So, if $(x, y) \in R$, then x will be related to the neighborhood y. The same shall hold for y in relation to x. Then $A_r{}^{(j)}(x) = E(x, a_r{}^{(j)})$ and $B_r(x) = E'(y, b_r)$ can be seen as fuzzy sets of values that are close to $a_r{}^{(j)}$ and b_r, respectively. Naturally, $\forall r = 1, \ldots, k : R\left(a_r^{(1)}, \ldots, a_r^{(p)}), b_r\right) = 1$. The user thus only needs to define reasonable similarity relations E_j and E' for each input ξ_j and the output η, respectively. Then, using the extensionality properties of R, one gets

$$R(x^{(1)}, \ldots, x^{(p)}, y) \geq \max_{r=1,\ldots,k} \otimes\left(A_r^{(1)}(x^{(1)}), \ldots, A_r^{(p)}(x^{(p)}), A_r(y)\right).$$

If we use the t-norm $\otimes = \min$, then Mamdani's approach to compute the fuzzy output is obtained. In (Boixader and Jacas 1998; Klawonn and Castro 1995) indistinguishability or similarity is expressed as link between the extensionality property and fuzzy equivalence relations. Fuzzy interpolation can be also seen as logical inference given fuzzy information coming from an vaguely known function (Klawonn and Novák 1996). Likewise, in Sudkamp (1993) fuzzy rules are obtained from set of pairs (a_i, b_i) and similarity relations on X and Y.

Takagi-Sugeno Controller

Takagi-Sugeno controllers (Takagi and Sugeno 1985) can be seen as modification of Mamdani-Assilian controllers. For both controllers, we need to specify fuzzy partitions of the input domains. However, no fuzzy partition of the output domain is needed since the rules R_r for $r = 1, \ldots, k$ are given as

$$R_r : \textbf{if } \xi_1 \textbf{is } A^{(1)}_{i_{1,r}} \textbf{ and} \ldots \textbf{and } \xi_n \textbf{ is } A^{(n)}_{i_{n,r}} \textbf{ then } \eta = f_r(\xi_1, \ldots, \xi_n).$$

Usually the functions f_r are linear, i.e., $f_r(\mathbf{x}) = a_r^{(0)} + \sum_{i=1}^{n} a_r^{(i)} x^{(i)}$.

Again, the decision logic determines the degree of applicability α_r of each premise using (16.7). These degrees are directly used to determine a crisp control value

$$\eta = \frac{\sum_{r=1}^{k} \alpha_r \cdot f_r(\mathbf{x})}{\sum_{r=1}^{k} \alpha_r}$$

which is a weighted sum over all rules' outputs. Hence, the defuzzification is omitted for that type of controller.

Approximate Reasoning

So far, we have treated the linguistic rules as associations of an n-dimensional fuzzy input point with one fuzzy output. This makes sense for control applications where each rule defines an operating point of the system to be controlled. Another way to interpret a fuzzy controller is to fuzzy constrain the control function by the fuzzy rules. This can be done by interpreting the inference process as approximate reasoning. In classical reasoning, tautologies/inference rules are used for deductive inferences of crisp conclusions from *crisp* propositions. Approximate reasoning can be seen as generalization of classical reasoning applied to *fuzzy* propositions. In (Zadeh 1973), first approaches have been developed to generalize approximate reasoning to fuzzy sets. In (Zadeh 1979, 1983), this methodology is explained in more detail. Using possibility distributions to represent incomplete knowledge helps to understand the mention techniques.

Whereas fuzzy set theory is closely associated with vague concepts, the application of possibility theory (Dubois and Prade 1988) relates to the imperfect description of an existing element x_0 in a set $A \subseteq X$. Possibility theory can be seen as counterpart to probability theory. In order to describe a possibility distribution $\Pi : 2^X \rightarrow [0, 1]$, the following axioms are used:

$$\begin{aligned}
&\Pi(\emptyset) = 0,\\
&\Pi(A) \leq \Pi(B) \text{ if } A \subseteq B \text{ and}\\
&\Pi(A \cup B) = \max\{\Pi(A), \Pi(B)\} \text{ for all } A, B \subset X.
\end{aligned}$$

$\Pi(A) = 1$ means that $x_0 \in A$ is unconditional possible. If $\Pi(A) = 0$ then it is impossible that $x_0 \in A$. In Zadeh (1978), uncertainty about x_0 is modeled by the possibility measure $\Pi : 2^\Omega \rightarrow [0, 1], \Pi(A) = \sup\{\mu(x) \mid x \in A\}$ when a fuzzy set $\mu : x \rightarrow [0, 1]$ is given as only description of x_0. For this special case the possibility measure is given by the possibility degrees of the singletons, i.e., $\Pi(\{x\}) = \mu(x)$.

For simplicity consider one-dimensional input and output spaces, respectively. Here, the choice of an appropriate two-dimensional possibility distribution is crucial. The rule

$$R : \textbf{if } \xi \textbf{ is } A \textbf{ then } \eta \textbf{ is } B$$

that associates the input fuzzy set μ_A with the output fuzzy set μ_B is modeled by a possibility distribution

$$\pi_{X,Y}(x, y) = I(\mu_A(x), \mu_B(y))$$

whereas I is an implication of a multivalued logic. Hence $\mu_B = \mu_A \circ \pi_{X,Y}$ where $\pi_{X,Y}$ is a fuzzy relation on $X \times Y$. The composition of a fuzzy set μ with a fuzzy relation π is defined by

$$\mu \circ \pi : Y \rightarrow [0, 1], \quad y \mapsto \sup_{x \in X} \{\min\{\mu(x), \pi(x, y)\}\}.$$

This is clearly a fuzzification of the composition $\circ$ of two crisp sets $M \subseteq X$ and $R \subseteq X \times Y$, i.e.,

$$M \circ R \overset{\text{def}}{=} \{y \in Y \mid \exists x \in X : (x \in M \wedge (x, y) \in R)\} \subseteq Y.$$

The task in fuzzy control based on such relational equations is to find a fuzzy relation π that fulfills all equations $\mu_{B_r} = \mu_{A_r} \circ \pi$ for every rule R_r with $r = 1, \ldots, k$. If multiple inputs $X_1, \ldots, X_n$ are used, then μ_A is defined on the product space $X = X_1$

$\times \ldots \times X_n$ as in (16.7). For each of the k relational equations, the *Gödel relation* is determined by

$$(x, y) \in \pi^G_{X,Y} \Longleftrightarrow (x \in \mu_A \to y \in \mu_B)$$

where the implication → is evaluated by the Gödel implication (see section "Fuzzy Logic as Many-Valued Logic"). Thus a linguistic rule can be seen as gradual rule 'The more μ_A, the more μ_B' which constrains π by the inequality

$$\min(\mu_A(x), \pi(x, y)) \leq \mu_B(y)$$

for all $(x, y) \in X \times Y$. Theoretically, different fuzzy implications could be used to describe π. However, several reasons can be found in favor for I_G, e.g., Dubois and Prade (1985, 1992).

If the system of relational equations $\mu_{B_r} = \mu_{A_r} \circ \pi$ for $r = 1, \ldots, k$ is solvable, then

$$\pi^G = \bigcap_{r=1}^{k} \pi^G_r(\mu_{A_r}(x), \mu_{B_r}(y))$$

is a solution with $\cap$ being the minimum t-norm. At the same time this is the greatest solution. We can say that the relation $\prod(\{(x, y)\}) \overset{\text{def}}{=} \pi(x, y)$ gives an estimate whether it is *possible* that input tuple x is assigned to output value y. So, the set of *conjunctive* rules imposes *soft constraints* on the control function φ. In practice, these constraints may lead to contradictions if narrow output fuzzy sets with overlapping input fuzzy sets are used. Thus the controller would output the empty fuzzy set, i.e., no solution. It is therefore reasonable to define rather narrow fuzzy sets for the input variables and rather broader fuzzy sets for the output.

Success of Mamdani Control in Automobile Industry

In the 1990s many real-world control applications have been greatly solved using Mamdani's approach. Among them are many control problems in the industrial automobile field. The number of publications, however, is really low. Two control applications at Volkswagen AG successfully use Mamdani's approach, i.e., the engine idle speed control and the shift-point determination of an automatic transmission (Schröder et al. 1997). The idle speed controller is based on similarity relations (see section "Mamdani-Assilian Controller"). This helps to view the control function as interpolation of a point-wise known function. The shift-point determination continuously adapts the gearshift schedule between two extremes, i.e., economic and sporting. A sport factor is computed to individually adapt the gearshift movements of a driver.

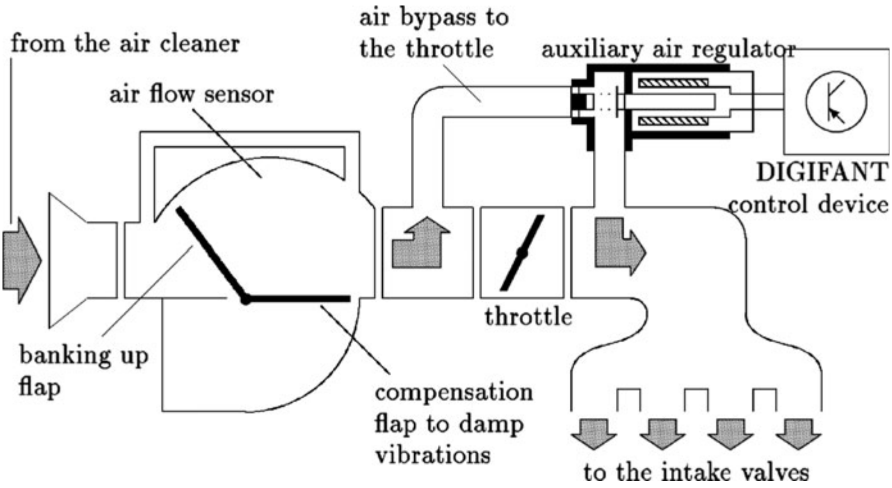


Fig. 16.3 Principle of the engine idle speed control

Engine Idle Speed Control

The task is to control the idle speed of a spark ignition engine. One way is a volumetric control where an auxiliary air regulator alters the cross-section of a bypass to the throttle. This is depicted in Fig. 16.3.

The pulse width of the auxiliary air regulator is changed by the controller. If there is a drop in the number of revolutions, then the controller forces the auxiliary air regulator to increase the bypass cross-section. The air flow sensor measures the increased air flow rate and thus notifies the controller. The new quantity for the fuel injection must be computed. Due to a higher air flow rate, the engine yields more torque. This again results in a higher number of revolutions which could be reduced analogously by decreasing the bypass cross-section.

Both fuel consumption and pollutant emissions should be ultimately reduced. This can be reached by slowing down the idle speed. However, a switching on of certain automobile facilities, e.g., air-conditioning system, forces the number of revolutions to drop. Hence the controller must be very flexible. More problems involved in this control application can be found in Schröder et al. (1997).

Due to this motivating problem, a Mamdani fuzzy controller was developed based on similarity relations. The resulting fuzzy controller was easier to design and showed an improved control behavior compared to classical control approaches. Similarity relations to represent indistinguishability or similarity of points within a certain vicinity seems to be a natural modeling way for engineers.

In fact, indistinguishability is not produced by measurement errors or deviations. It just expresses that arbitrary precision is not necessary to control a system. A control expert must thus specify a set of k input-output tuples $((x_r^{(1)}, \ldots, x_r^{(p)}), y_r)$. For each $r = 1, \ldots, k$, the output value y_r seems appropriate for the input $(x_r^{(1)}, \ldots, x_r^{(p)})$. So, the human expert defines the partial control function φ_0.

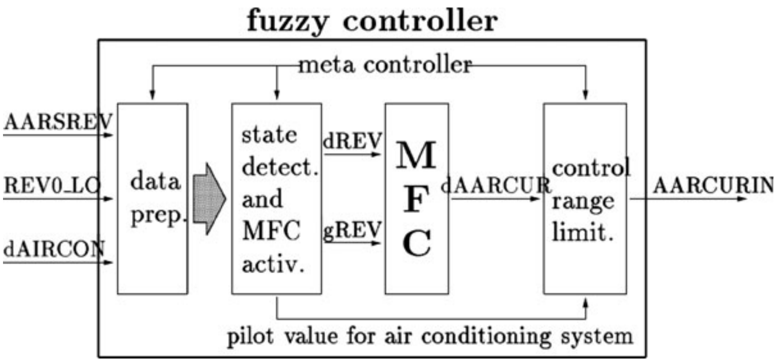


Fig. 16.4 Structure of the fuzzy controller

In the 1990s the question to be answered was to compute a suitable output value for an arbitrary input given specified similarity relations and φ_0 (Schröder et al. 1997). Using the extensionality properties defined in (16.11), one obtains Mamdani's fuzzy output directly by computing the extensional hull of φ_0 given the similarity relations. The partial control function φ_0 can thus be reinterpreted as k control rules of the form:

$$R_r: \ \textbf{if } \xi_1 \textbf{ is } \textit{approximately } x_r^{(1)} \textbf{ and } \ldots \textbf{ and } \xi_p \textbf{ is } \textit{approximately } x_r^{(p)} \\ \textbf{then } \eta \textbf{ is } \textit{approximately } y_r.$$

A more profound theoretical analysis of this approach can be found in Klawonn et al. (1995).

To control the engine idle speed controller, two input variables are needed:

1. The deviation dREV [rpm] of the number of revolutions to the set value, and
2. The gradient gREV [rpm] of the number of revolutions between two ignitions.

The only output variable is the change of current dAARCUR for the auxiliary air regulator. The controller is shown in Fig. 16.4.

The knowledge to control the engine idle speed controller was extracted by measurement data obtained from idle speed experiments. The partial control mapping $\varphi_0 : X_{\text{(dREV)}} \times X_{\text{(gREV)}} \to Y_{\text{(dAARCUR)}}$ has been specified as in Table 16.1 (left-hand side).

Using a similarity relation and φ_0, the fuzzy controller was defined. Its induced control surface is shown in Fig. 16.5 as a grid of supporting points. The center of area (COA) method has been used for defuzzification. To obtain the corresponding Mamdani fuzzy controller, each point of φ_0 was associated with a linguistic term, e.g., negative big (nb), negative medium (nm), negative small (ns), approximately zero (az), and so on. The obtained fuzzy partitions of all three variables are shown in Figs. 16.6–16.8, respectively. The partial mapping φ_0 was translated into linguistic rules of the form

$$\textbf{if } \text{dREV} \textbf{ is } A \textbf{ and } \text{gREV} \textbf{ is } B \textbf{ then } \text{dAARCUR} \textbf{ is } C.$$

Table 1.1 The partial control mapping ∏ 0 (left-hand side) and its corresponding fuzzy rule base (right-hand side).

		gREV						
		–40	–6	–3	0	3	6	40
	–70	20	15	15	10	10	5	5
	–50	20	15	10	10	5	5	0
	–30	15	10	5	5	5	0	0
dREV	0	5	5	0	0	0	–10	–5
	30	0	0	0	–5	–5	–10	–15
	50	0	–5	–5	–10	–15	–15	–20
	70	–5	–5	–10	–15	15	15	15

		gREV						
		nb	nm	ns	az	ps	pm	pb
	nb	ph	pb	pb	pm	pm	ps	ps
	nm	ph	pb	pm	pm	ps	ps	az
	ns	pb	pm	ps	ps	az	az	az
dREV	az	ps	ps	az	az	az	nm	ns
	ps	az	az	az	ns	ns	nm	nb
	pm	az	ns	ns	ns	nb	nb	nh
	pb	ns	ns	nm	nb	nb	nb	nh

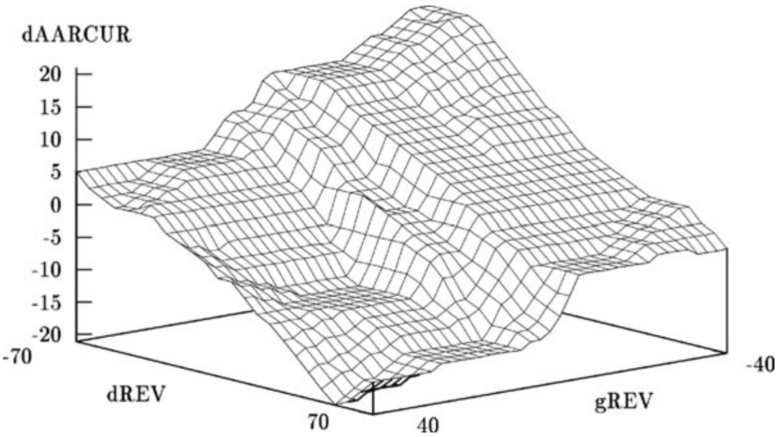
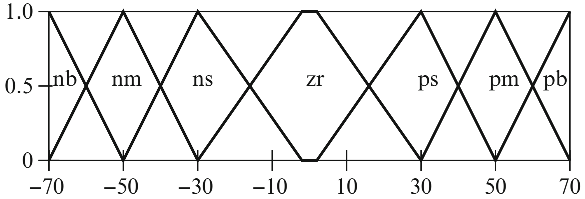


Fig. 16.5 Performance characteristics

Fig. 16.6 Deviation dREV of the number of revolutions



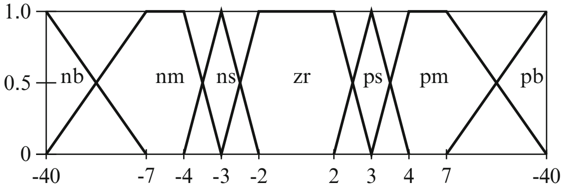


Fig. 16.7 Gradient gREV of the number of revolutions

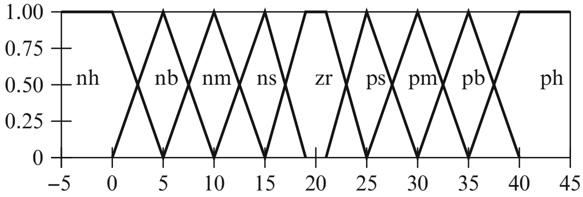


Fig. 16.8 Change of current dAARCUR for the auxiliary air regulator

The complete set of rules is given on the right-hand side of Table 16.1.

In (Klawonn et al. 1995; Schröder et al. 1997) the Mamdani fuzzy controller shows a very smooth control behavior compared to its serial counterpart. Furthermore the fuzzy controller reaches the desired set point precisely and fast. Its behavior is robust even with slowly increasing load. Thus the number of revolutions does not lead to any vibration even after extreme changes of load occur.

Flowing Shift-Point Determination

Conventional automatic transmissions select gears based on so-called gearshift diagrams. Here, the gearshift simply depends on the accelerator position and the velocity. A lagging between up and down shift avoids oscillating gearshift when the velocity varies slightly, e.g., during stop-and-go traffic. For a standardized behavior, a fixed diagram works well. Until 1994, the Volkswagen gear box had two different types of gearshift diagrams, i.e., economic "ECO" and sporting "SPORT". An economic gearshift diagram switches gears at a low number of revolutions to reduce the fuel consumption. A sporting one leads to gearshifts at a higher number of revolutions. Since 1991 it was a research issue at Volkswagen AG to develop an individual adaption of shift-points. No additional sensors should be used to observe the driver.

The idea was that the car "observes" the driver (Schröder et al. 1997) and classifies him or her into calm, normal, sportive (assigning a sport factor $\in [0, 1]$), or nervous (to calm down the driver). A test car from Volkswagen was operated by many different drivers. These people were classified by a human expert (passenger). Simultaneously, 14 attributes were continuously measured during test drives. Among them were variables like the velocity of the car, the position of the acceleration pedal, the speed of the acceleration pedal, the kick down, or the steering wheel angle.

The final Mamdani controller was based on four input variables and one output. The basic structure of the controller is shown in Fig. 16.9. In total, 7 rules could be

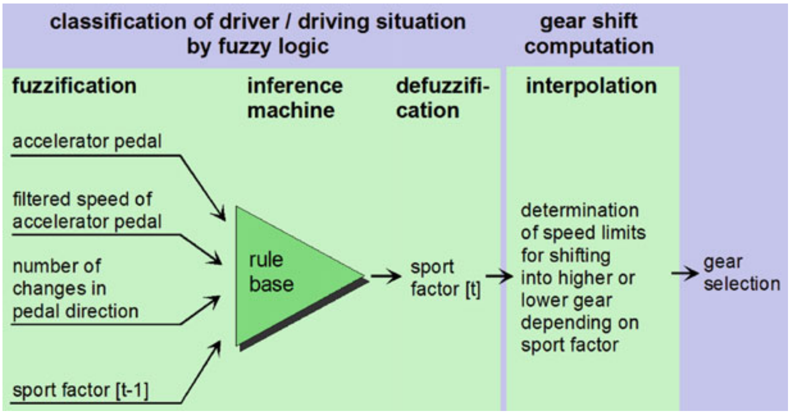

classification of driver / driving situation by fuzzy logic
gear shift computation
fuzzification
inference machine
defuzzifi-cation
interpolation
accelerator pedal
filtered speed of accelerator pedal
number of changes in pedal direction
sport factor [t-1]
rule base
sport factor [t]
determination of speed limits for shifting into higher or lower gear depending on sport factor
gear selection

Fig. 16.9 Flowing shift-point determination with fuzzy logic

identified at which the antecedent consists of up to 4 clauses. The program was highly optimized: It used 24 Byte RAM and 702 Byte ROM, i.e., less than 1 KB. The runtime was 80 ms which means that 12 times per second a new sport factor was assigned. The controller is in series since January 1995. It shows an excellent performance.

Fuzzy Logic and Knowledge Discovery in Databases

Knowledge discovery in databases (KDD) tries to inspect, clean, transform and model *data* in large databases in order to find useful information or support decision making. Ultimately, one tries to formulate knowledge based on pieces of information that have been discovered in databases. A single datum may describe the condition of a certain object. It carries only information if there are at least two different states of the condition. A datum might be seen as the realization of a certain variable of a universe. There are different representations of a datum as it has been measured, i.e., nominal, ordinal, interval or ratio (Stevens 1946).

The KDD process is usually performed in four stages. At the first stage, the data are valuated and examined w.r.t. simple and essential characteristics, e.g., analysis of frequency, reliability test, runaway, credibility. The second stage comprises pattern matching or the grouping of observations. Usually transformations are performed with the goal to find structures within data. At that stage, exploratory data analysis is performed to examine the data without a previously chosen mathematical model. At the third level, data are analyzed w.r.t. one or more mathematical models. These models can be either qualitative or quantitative. The former one is the formation relating to additional characteristics expressed by quality, e.g., introduction of the term of similarity for cluster analysis. The latter type of models tries to recognize functional relations, e.g., an approximation of regression analysis.

At the fourth level, conclusions from the whole process are drawn and evaluated. Also, future or missing values might be predicted. Sources of data may be combined by, e.g., data fusion. In general, the learned models are revised at that stage.

If data are vague, imprecise or inconsistent, the application of fuzzy logic to KDD might improve results. Usually common data are analyzed by fuzzy methods whereas some researchers also analyze fuzzy data. The most prominent approach to fuzzy data analysis is fuzzy clustering that is introduced in section "Fuzzy Clustering". Its successfulness in KDD might come from the fact that human beings do not group objects based on crisp labels. We rather use some kind of fuzzy terms to cluster things, e.g., into the group of tall people. Many everyday decisions are fuzzy and human beings are able to handle that. Therefore an appropriate answer to the following question is naturally important: How can a computer learn fuzzy rules from data to explain or support decisions like people do? We describe some general approaches to generate fuzzy rules from data in section "Fuzzy Rule Generation".

Fuzzy Clustering

Clustering is an unsupervised learning task that tries to divide data s.t.

- Objects belonging to the same cluster are as similar as possible, and
- Objects belonging to different clusters are as dissimilar as possible.

Similarity is normally measured in terms of a distance function. The smaller the distance, the more similar two data tuples. Here, we assume that every data tuple is an element of the n-dimensional Euclidean space IR^n.

Definition 1 (Distance function). *The mapping d : IR$^n\times$IR$^n\rightarrow$ [0,∞) is a distance function if it satisfies the following conditions for all* **x,y,z** $\in$ IR $^{\mathrm{n}}$:

1.	$d(\mathbf{x},\mathbf{y}) = 0 \Leftrightarrow \mathbf{x} = \mathbf{y}$	(identity),
2.	$d(\mathbf{x},\mathbf{y}) = d(\mathbf{y},\mathbf{x})$	(symmetry),
3.	$d(\mathbf{x},\mathbf{z}) \leq d(\mathbf{x},\mathbf{y}) + d(\mathbf{y},\mathbf{z})$	(triangle inequality).

Henceforth we only focus on partitioning algorithms, i.e., given a number $c \in$ IN, find the best partition of data into c groups. This is fundamentally different from hierarchical clustering techniques where data are organized in a nested sequence of groups (e.g., dendrograms). Usually the true number of clusters is unknown which makes it hard to use partitioning methods. To further specify, we concentrate on *prototype-based clustering* algorithms where clusters are represented by prototypes $C_i, i = 1,\dots,c$. The prototypes shall capture the structure/distribution of data in each cluster. They are constructed by clustering algorithms. For simplicity, consider cluster prototypes C_i which are solely represented by the cluster centers $\mathbf{c}_i$. Furthermore, the distance measure d is based on the inner product, e.g., the Euclidean distance

$$d(\mathbf{x},\mathbf{y}) = \|\mathbf{x}-\mathbf{y}\|_2 = \sqrt{\sum_{i=1}^{n}(x^{(i)}-y^{(i)})^2}.$$

Every prototype-based clustering algorithm is based on an objective function J that quantifies the goodness of the cluster model. J must be minimized to obtain optimal clusters. The algorithms determine the best decomposition by minimizing J.

The simplest algorithm is called hard c-means or *k-means* clustering. Here, each data point $\mathbf{x}_j$ in dataset $\mathcal{X} = \{\mathbf{x}_1,\ldots,\mathbf{x}_m\}, \mathcal{X} \subseteq \mathrm{IR}^n$ is assigned to exactly one cluster $\Gamma_i \subset \mathcal{X}$. The set of clusters $\mathit{\Gamma} = \{\mathit{\Gamma}_1,\ldots,\mathit{\Gamma}_c\}$ must be an exhaustive partition of $\mathcal{X}$ into c non-empty and pairwise disjoint subsets $\mathit{\Gamma}_i$, $1 < i < c$. The data partition is optimal when the sum of squared distances between cluster centers and data points assigned to them is minimal. The clusters should be as homogeneous as possible. The objective function of hard c-means is thus

$$J_h(X,U_h,C) = \sum_{i=1}^{c}\sum_{j=1}^{m} u_{ij}d_{ij}^2 \tag{16.12}$$

whereas d_{ij} is the distance between $\mathbf{c}_i$ and $\mathbf{x}_j$, $U = u_{ij} \in \{0,1\}_{c\times m}$ is called *partition matrix* with

$$u_{ij} = \begin{cases} 1 & \text{if } \mathbf{x}_j \in \Gamma_i, \\ 0 & \text{otherwise.} \end{cases}$$

Equation 16.12 is minimized subject to the following two constraints: Each data point is assigned exactly to one cluster, i.e.,

$$\sum_{i=1}^{c} u_{ij} = 1, \quad \forall j \in \{1,\ldots,m\}. \tag{16.13}$$

Every cluster must contain at least one data point, i.e.,

$$\sum_{j=1}^{m} u_{ij} > 0, \quad \forall i \in \{1,\ldots,c\}. \tag{16.14}$$

J_h depends on both c and the assignment U of data points to the clusters. Finding the parameters that minimize J_h is NP-hard. Therefore J_h is minimized by *alternating optimization* (AO). The parameters to optimize are split into two groups. One group is optimized holding the other group fixed (and vice versa). An iterative update scheme is repeated until the algorithm converges. It cannot be guaranteed that a global optimum will be reached. Hence, the algorithm may get stuck in a local minimum. The AO scheme for hard c-means first choses c initial $\mathbf{c}_i$, e.g., by randomly picking c data points from $\mathcal{X}$. Then, C is fixed and U is determined that minimizes J_h. This is done by assigning each data point to its closest cluster center, i.e.,

$$u_{ij} = \begin{cases} 1 & \text{if } i = \arg\min_{k=1}^{c} d_{kj}, \\ 0 & \text{otherwise.} \end{cases}$$

After that U is fixed and $\mathbf{c}_i$ are updated as the mean of all $\mathbf{x}_j$ assigned to them. The mean minimizes the sum of square distances in J_h, i.e.,

$$\mathbf{c}_i = \frac{\sum_{j=1}^{m} u_{ij}\mathbf{x}_j}{\sum_{j=1}^{m} u_{ij}}.$$

Finally, both steps are repeated until no change in C or U can be observed.

The hard c-means algorithm tends to get stuck in local minimum. It is therefore necessary to conduct several runs with different initializations (Duda and Hart 1973). The best result of many clusterings can be chosen based on the value of J_h. The crisp memberships $u_{ij} \in \{0, 1\}$ prohibit ambiguous assignments. When clusters are badly delineated or overlapping, relaxing this requirement is needed. This can be achieved using fuzzy clustering.

Fuzzy clustering algorithms allow gradual memberships of data points to a cluster in $[0, 1]$. A data point can thus belong to more than one cluster. Consequently, the membership degrees offer finer degrees of detail and express how ambiguously $\mathbf{x}_j$ should belong to Γ_i. The clusters Γ_i have been classical subsets so far. Now, they are represented by fuzzy sets μ_{Γ_i} of $\mathcal{X}$. Instantly, the cluster assignment u_{ij} is the membership degree of $\mathbf{x}_j$ to Γ_i s.t. $u_{ij} = \mu_{\Gamma_i}(\mathbf{x}_j) \in [0, 1]$. Thence, a fuzzy label vector $\mathbf{u} = (u_{1j}, \ldots, u_{cj})^T$ is linked to each $\mathbf{x}_j$. The matrix $U = (u_{ij}) = (\mathbf{u}_1, \ldots, \mathbf{u}_m)$ is then called *fuzzy partition matrix*. Two types of fuzzy cluster partitions are known, i.e., *probabilistic* and possibilistic. The differ in the constraints they place on the membership degrees. For a *probabilistic cluster partition*, the constraints expressed by (16.13) and (16.14) must hold. So, no cluster can contain the full membership of all data points. Also, the membership degrees for a given datum resemble the probabilities of being member of a corresponding cluster. A *possibilistic cluster partition* only needs to fulfill the constraint (16.13). Here, we only focus on the former type of cluster partition. Algorithms based on the latter one can be found in Höppner et al. (1999).

In order to handle fuzzy membership assignments, we must minimize the objective function

$$J_f(X, U_h, C) = \sum_{i=1}^{c} \sum_{j=1}^{m} u_{ij}^{w} d_{ij}^{2}$$

subject to (16.13) and (16.14). The parameter $w \in \mathrm{IR}$ with $w > 1$ is called *fuzzifier*. The value of w determines the "fuzziness" of the grouping. For $w = 1$ (i.e., $J_h = J_f$), the assignments remain hard. Only fuzzifiers $w > 1$ lead to fuzzy memberships (Bezdek 1973). Thus the clusters become softer/harder with higher/lower w. Usually w is set to 2 in most applications. The function J_f is alternately optimized, i.e., first optimizing U for fixed cluster parameters $U_\tau = j_U(C_{\tau-1})$, then optimizing C for fixed membership degrees $C_\tau = j_C(U_\tau)$. The update formulas can be determined by setting the derivative of J_f w.r.t. U and C to zero. The resulting equations form the *fuzzy c-means* (FCM) algorithm. The membership degrees are chosen according to Bezdek (1981)

$$u_{ij} = \frac{1}{\sum_{k=1}^{c} \left(\frac{d_{ij}^2}{d_{kj}^2} \right)^{\frac{1}{w-1}}} = \frac{d_{ij}^{\frac{2}{1-w}}}{\sum_{k=1}^{c} d_{kj}^{\frac{2}{1-w}}}$$

which is independent of the chosen distance measure. For the basic FCM model With the second step of the AO scheme, the derivations of J_f w.r.t. the centers yield (Bezdek 1981)

$$\mathbf{c}_i = \frac{\sum_{j=1}^{m} u_{ij}^m \mathbf{x}_j}{\sum_{j=1}^{m} u_{ij}^m}.$$

Like hard c-means, FCM can be initialized with randomly placed cluster centers. Updating in the AO scheme can be stopped if the number of iterations τ exceeds some predefined $\tau_{\max}$ or if changes in the prototypes are smaller than some termination accuracy. FCM is stable and robust. Compared to hard c-means, it is quite insensitive to the initialization and not likely to get stuck in a local minimum. FCM converges in a saddle point or minimum (but not in a maximum) Bezdek (1981). Further fuzzy clustering algorithms, distance functions variants and applications can be found in Bezdek et al. (1999) and Höppner et al. (1999).

Fuzzy Rule Generation

The automatic generation of linguistic rules plays an important role in many applications, e.g., classification (Kuncheva 2000; Nauck and Kruse 1997), regression (Dickerson and Kosko 1996; Nauck and Kruse 1999; Wang and Mendel 1992), control engineering (Klawonn et al. 1995; Klawonn and Kruse 1993, 1995, 1997), image processing (Bezdek et al. 1999; Höppner et al. 1999). In fuzzy data analysis, we are interested in learning fuzzy rules from observations using fuzzy methods, e.g., FCM.

Before we talk about the generation of linguistic rules from fuzzy clustering, let us briefly mention the some other methods based on fuzzy logic. Grid-based approaches define fixed fuzzy partitions for every variable. Every cell in that multidimensional grid may correspond to one rule (Wang and Mendel 1992). Most well-known are hybrid methods to induce fuzzy rules. Therefore a fuzzy system is combined with computational intelligence techniques. For instance, *evolutionary algorithms* are used for guided searching the space of possible rule bases (Cordón et al. 2004). *Neuro-fuzzy systems* use learning methods of artificial neural network (e.g., backpropagation) to tune parameters of a network that can be directly understood as a fuzzy system (Nauck et al. 1997). Standard rule generation methods have been fuzzified as well (e.g., separate-and-conquer rule learning (Hühn and Hüllermeier 2009), decision trees (Olaru and Wehenkel 2003), support vector machines (Moewes and Kruse 2008).

Here, we will restrict ourselves to FCM for fuzzy rule generation. Consider again the input space $X \subset \mathrm{IR}^n$ and the output space $Y \subset \mathrm{IR}$. We observe m patterns $(\mathbf{x}_j, y_j) \in S \subseteq X \times Y$ where $j = 1, \ldots, m$. Running FCM on that dataset S leads to c cluster prototypes $\mathbf{c}_i = (c_i^{(1)}, \ldots, c_i^{(n)}, c_i^{(y)})$ with $i = 1, \ldots, c$ that can be seen as concatenation of both the input values $c_i^{(j)}, j = 1, \ldots, n$ and the output value $c_i^{(y)}$. Thus every prototype represents one linguistic rule

$$R_i: \text{ if } x \text{ is close to } \left(c_i^{(1)}, \ldots, c_i^{(n)}\right) \text{ then } y \text{ is close to } c_i^{(y)}.$$

Using the membership degrees U, we can rewrite these rules as

$$R_i: \text{ if } \mathbf{u}_i^{\mathbf{x}}(\mathbf{x}) \text{ then } u_i^y(y). \tag{16.15}$$

The only problem is that FCM returns the membership degrees $\mathbf{u}_i(\mathbf{x}, y)$ of the product space $X \times Y$. To obtain rules like (16.15), we must *project* $\mathbf{u}_i$ onto $\mathbf{u}_i^{\mathbf{x}}$ and u_i^y. If $\mathbf{x}$ and y are restricted to $[\mathbf{x}_{\min}, \mathbf{x}_{\max}]$ and $[y_{\min}, y_{\max}]$, respectively, the projections are given by

$$\begin{aligned}
\mathbf{u}_i^{\mathbf{x}}(\mathbf{x}) &= \sup_{y \in [y_{\min}, y_{\max}]} \mathbf{u}_i(\mathbf{x}, y), \\
u_i^y(y) &= \sup_{x \in [\mathbf{x}_{\min}, \mathbf{y}_{\max}]} \mathbf{u}_i(\mathbf{x}, y).
\end{aligned}$$

We can also project $\mathbf{u}_i$ onto each single input variable $X_1, \ldots, X_n$ by

$$u_{ik}(x^{(k)}) = \sup_{\mathbf{x}^{(\neg k)} \in [\mathbf{x}_{\min}^{(\neg k)}, \mathbf{x}_{\max}^{(\neg k)}]} \mathbf{u}_i^{\mathbf{x}}(\mathbf{x})$$

for $k = 1, \ldots, n$ where as $\mathbf{x}^{(\neg k)} \overset{\text{def}}{=} (x^{(1)}, \ldots, x^{(k-1)}, x^{(k+1)}, \ldots, x^{(n)})$. We may thus write (16.15) in form of a *Mamdani-Assilian rule* (16.6) as

$$R_i: \text{ if } \bigwedge_{k=1}^{n} u_{ik}(x^{(k)}) \text{ then } u_i^y(y). \tag{16.16}$$

For one rule, the output value of an unseen input $\mathbf{x} \in \mathrm{IR}^n$ will be equivalent to (16.7) if the minimum t-norm is used as conjunction $\wedge$. The overall output of the complete rule base is given by a disjunction $\vee$ of all rule outputs (cf. (16.9) if $\vee$ is the t-conorm maximum).

A crisp output can then again be computed by defuzzification, e.g., using the COG method (16.10). Since this computation is rather costly, the output membership functions u_i^y are commonly be replaced by singletons, i.e.,

$$u_i^y(y) = \begin{cases} 1 & \text{if } y = c_i^{(y)}, \\ 0 & \text{otherwise.} \end{cases}$$

Since each rule consequent comprise the component $c_i^{(y)}$ of the cluster prototype, we can rewrite (16.16) as *Sugeno-Yasukawa rule* (Sugeno and Yasukawa 1993)

$$R_i: \text{ if } \bigwedge_{k=1}^{n} u_{ik}(x^{(k)}) \text{ then } y = c_i^{(y)}.$$

These rules strongly resemble the neurons of an RBF network. This will become clear if every membership function is Gaussian, i.e.,

$$\mathbf{u}_i^{\mathbf{x}}(\mathbf{x}) = \exp\left(\frac{\mathbf{x} - \mu_i}{\sigma_i}\right)^2,$$

and if there are normalized, i.e., $\sum_{i=1}^{c} \mathbf{u}_i^{\mathbf{x}}(\mathbf{x}) = 1$ for all $\mathbf{x} \in \mathrm{IR}^n$. This link is used in neuro-fuzzy systems for both training fuzzy rules with backpropagation and initializing RBF networks with fuzzy rules (Nauck and Kruse 1997).

Transfer Passenger Analysis Based on FCM

The German Aerospace Center (DLR) developed a macroscopic passenger flow model for simulating passenger movements on airport's land side. For the passenger movements in terminal areas, probabilistic distribution functions are used today. In (Keller and Kruse 2002), the goal was to build a fuzzy rule base describing the transfer passenger amount between aircrafts. These rules could be used to improve the macroscopic simulation. The key idea was to find the rules based on FCM. The following attributes of passengers were used to for analysis:

- The maximal amount of passengers in a certain aircraft (depending on the type of the aircraft)
- The distance between the airport of departure and the airport of destination (in three categories: short-, medium-, and long-haul)
- The time of departure
- The percentage of transfer passengers in the aircraft

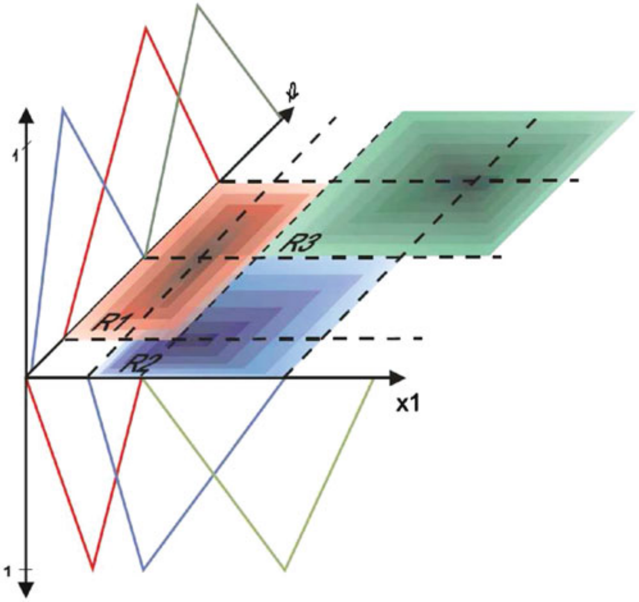


Fig. 16.10 Fuzzy rules and induced vague areas

The number of clusters were determined by validity measures (Höppner et al. 1999; Kruse et al. 2007) evaluating the whole partition of all data. The clustering was run for a varying number of clusters. The validity of the resulting partitions was compared based on the used measures.

An example of resulting fuzzy clusters are shown in Fig. 16.10. Every fuzzy cluster corresponds to one fuzzy rule. The color intensity indicates the firing strength of a specific rule. The vague areas are the fuzzy clusters whereas the color intensity indicates the membership degree. The tips of the fuzzy partitions are obtained in every domain by projections of the multidimensional cluster centers (as explained before in section "Fuzzy Rule Generation").

The fuzzy rules obtained by FCM were simplified through several steps. First, similar fuzzy sets were combined to one fuzzy set. Fuzzy sets similar to the universal fuzzy set were removed. Fuzzy rules with the same input clauses were either combined if they also shared the same output clauses or else they were removed from the rule base. Finally, around five rules could be obtained from FCM. Among them were the two following rules: If an aircraft with a relatively small amount of maximal passengers (80–200) has a short- or medium-haul destination departing late at night, then usually this flight has a high amount of transfer passengers (80–90%). If a flight with a medium-haul destination and a small aircraft (about 150 passengers) starts about noon, then it carries a relatively high amount of transfer passengers (ca. 70%). We refer to Keller and Kruse (2002) for more details about this real-world application.

Acknowledgment R. Belohlavek was supported by the ESF project No. CZ.1.07/2.3.00/20.0059 (co-financed by the European Social Fund and the state budget of the Czech Republic).

References

Bezdek, J.C.: Fuzzy mathematics in pattern classification. PhD thesis, Cornell University, Itheca, NY, USA (1973)

Bezdek, J.C.: Pattern Recognition with Fuzzy Objective Function Algorithms. Kluwer Academic Publishers, Norwell, MA, USA (1981)

Bezdek, J.C., Keller, J., Krisnapuram, R., Pal, N.R.: Fuzzy Models and Algorithms for Pattern Recognition and Image Processing, *The Handbooks of Fuzzy Sets*, vol. 4. Kluwer Academic Publishers (1999)

Boixader, D., Jacas, J.: Extensionality based approximate reasoning. International Journal of Approximate Reasoning **19**(3–4), 221–230 (1998). DOI 10.1016/S0888-613X(98)00018-8

Belohlavek, R.: Fuzzy Relational Systems: Foundations and Principles, *IFSR International Series on Systems Science and Engineering*, vol. 20. Kluwer Academic Publishers (2002)

Belohlavek, R., Klir, G.J.: On Elkan's theorems: Clarifying their meaning via simple proofs: Research articles. International Journal of Intelligent Systems **22**, 203–207 (2007). DOI 10.1002/int.v22:2. ACM ID: 1190438

Belohlavek, R., Klir, G.J. (eds.): Concepts and Fuzzy Logic. MIT Press, Cambridge, MA, USA (2011)

Belohlavek, R., Vychodil, V.: Fuzzy Equational Logic, *Studies in Fuzziness and Soft Computing*, vol. 186. Springer (2005)

Belohlavek, R., Vychodil, V.: Attribute implications in a fuzzy setting. In: Formal Concept Analysis, *Lecture Notes in Computer Science*, vol. 3874, pp. 45–60. Springer-Verlag (2006)

Cheeseman, P.: An inquiry into computer understanding. Computational Intelligence **4**, 58–66 (1988)

Cheeseman, P.: Probabilistic versus fuzzy reasoning. In: L.N.K. Kanal, Lemmer (eds.) UAI '85: Proceedings of the First Annual Conference on Uncertainty in Artificial Intelligence. Elsevier, New York, NY, USA (1988)

Cignoli, R., Esteva, F., Godo, L., Torrens, A.: Basic fuzzy logic is the logic of continuous t-norms and their residua. Soft Computing **4**(2), 106–112 (2000). DOI 10.1007/s005000000044

Cordón, O., del Jesus, M.J., Herrera, F.: A proposal on reasoning methods in fuzzy rule-based classification systems. International Journal of Approximate Reasoning **20**(1), 21–45 (1999). DOI 10.1016/ S0888-613X(00)88942-2

Cordón, O., Gomide, F., Herrera, F., Hoffmann, F., Magdalena, L.: Ten years of genetic fuzzy systems: current framework and new trends. Fuzzy Sets and Systems **141**(1), 5–31 (2004). DOI 10.1016/S0165-0114(03) 00111-8

Dickerson, J.A., Kosko, B.: Fuzzy function approximation with ellipsoidal rules. IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics **26**(4), 542–560 (1996). DOI 10.1109/3477.517030

Dubois, D., Prade, H.: The generalized modus ponens under sup-min composition – a theoretical study. In: M.M. Gupta, A. Kandel, W. Bandler, J.B. Kiszka (eds.) Approximate Reasoning in Expert Systems, pp. 217–232. Elsevier Science Publisher B.V. (North-Holland), Amsterdam, Netherlands (1985)

Dubois, D., Prade, H.: Possibility Theory: An Approach to Computerized Processing of Uncertainty. Plenum Press, New York, NY, USA (1988)

Dubois, D., Prade, H.: Possibility theory as a basis for preference propagation in automated reasoning. In: 1992 IEEE International Conference on Fuzzy Systems, pp. 821–832. IEEE Press, New York, NY, USA (1992). DOI 10. 1109/FUZZY.1992.258765

Duda, R.O., Hart, P.E.: Pattern Classification and Scene Analysis. John Wiley & Sons, Ltd., New York, NY, USA (1973)

Elkan, C.: The paradoxical success of fuzzy logic. In: Proceedings of the 11th National Conference on Artificial Intelligence, pp. 698–703. AAAIPress/MIT Press, Cambridge, MA, USA (1993)

Elkan, C.: The paradoxical success of fuzzy logic. IEEE Expert: Intelligent Systems and Their Applications **9**, 3–8 (1994). DOI 10.1109/64.336150. ACM ID: 630036

Gerla, G.: Fuzzy Logic: Mathematical Tools for Approximate Reasoning. Kluwer Academic Publishers, Dordrecht, Netherlands (2001)

Goguen, J.A.: The logic of inexact concepts. Synthese **19**(3–4), 325–373 (1968). DOI 10.1007/BF00485654

Gottwald, S.: A Treatise on Many-Valued Logics, *Studies in Logic and Computation*, vol. 9. Research Studies Press, Baldock, Hertfordshire, England (2001)

Hájek, P.: Metamathematics of Fuzzy Logic, *Trends in Logic*, vol. 4. Kluwer Academic Publishers, Boston, MA, USA (1998)

Hájek, P.: What is mathematical fuzzy logic. Fuzzy Sets and Systems **157**(5), 597–603 (2006). DOI 10.1016/j.fss.2005.10.004

Höppner, F., Klawonn, F., Kruse, R., Runkler, T.: Fuzzy Cluster Analysis: Methods for Classification, Data Analysis and Image Recognition. John Wiley & Sons, Ltd., New York, NY, USA (1999)

Hühn, J.C., Hüllermeier, E.: FR3: a fuzzy rule learner for inducing reliable classifiers. IEEE Transactions on Fuzzy Systems **17**(1), 138–149 (2009). DOI 10.1109/TFUZZ.2008.2005490

Keller, A., Kruse, R.: Fuzzy rule generation for transfer passenger analysis. In: L. Wang, S.K. Halgamuge, X. Yao (eds.) Proceedings of the 1st International Conference on Fuzzy Systems and Knowledge Discovery (FSDK'02), pp. 667–671. Orchid Country Club, Singapore (2002)

Klawonn, F., Castro, J.L.: Similarity in fuzzy reasoning. Mathware & Soft Computing **2**(3), 197–228 (1995)

Klawonn, F., Gebhardt, J., Kruse, R.: Fuzzy control on the basis of equality relations with an example from idle speed control. IEEE Transactions on Fuzzy Systems **3**(3), 336–350 (1995). DOI 10.1109/91.413237

Klawonn, F., Kruse, R.: Equality relations as a basis for fuzzy control. Fuzzy Sets and Systems **54**(2), 147–156 (1993). DOI 10.1016/0165-0114(93) 90272-J

Klawonn, F., Kruse, R.: Automatic generation of fuzzy controllers by fuzzy clustering. In: 1995 IEEE International Conference on Systems, Man, and Cybernetics: Intelligent Systems for the 21st Century, vol. 3, pp. 2040–2045. IEEE Press, Vancouver, BC, Canada (1995). DOI 10.1109/ ICSMC.1995.538079

Klawonn, F., Kruse, R.: Constructing a fuzzy controller from data. Fuzzy Sets and Systems **85**(2), 177–193 (1997). DOI 10.1016/0165-0114(95)00350-9

Klawonn, F., Novák, V.: The relation between inference and interpolation in the framework of fuzzy systems. Fuzzy Sets and Systems **81**(3), 331–354 (1996). DOI 10.1016/0165-0114(96) 83710-9

Klement, E.P., Mesiar, R., Pap, E.: Triangular Norms, *Trends in Logic*, vol. 8. Kluwer Academic Publishers, Dordrecht, Netherlands (2000)

Klir, G.J.: Is there more to uncertainty than some probability theorists might have us belive? International Journal of General Systems **15**(4), 347–378 (1989). DOI 10.1080/03081078908935057

Klir, G.J., Yuan, B.: Fuzzy Sets and Fuzzy Logic: Theory and Applications. Prentice Hall, Upper Saddle River, NJ, USA (1995)

Kosko, B.: Fuzziness vs. probability. International Journal of General Systems **17**(2), 211–240 (1990). DOI 10.1080/03081079008935108

Kruse, R., Döring, C., Lesot, M.: Fundamentals of fuzzy clustering. In: J.V. de Oliveira, W. Pedrycz (eds.) Advances in Fuzzy Clustering and its Applications, pp. 3–30. John Wiley & Sons, Ltd., Chichester, UK (2007)

Kruse, R., Gebhardt, J., Klawonn, F.: Foundations of Fuzzy Systems. John Wiley & Sons Ltd, Chichester, UK (1994)

Kuncheva, L.I.: Fuzzy Classifier Design, *Studies in Fuzziness and Soft Computing*, vol. 49. Physica-Verlag, Heidelberg, New York (2000)

Lindley, D.V.: The probability approach to the treatment of uncertainty in artificial intelligence and expert systems. Statistical Science **2**(1), 17–24 (1987)

Loginov, V.I.: Probability treatment of zadeh membership functions and their use in pattern recognition. Engineering Cybernetics **4**(2), 68–69 (1966)

Mamdani, E.H., Assilian, S.: An experiment in linguistic synthesis with a fuzzy logic controller. International Journal of Man-Machine Studies **7**(1), 1–13 (1975). DOI 10.1016/S0020-7373 (75)80002-2

Moewes, C., Kruse, R.: Unification of fuzzy SVMs and rule extraction methods through imprecise domain knowledge. In: J.L. Verdegay, L. Magdalena, M. Ojeda-Aciego (eds.) Proceedings of the International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems (IPMU-08), pp. 1527–1534. Torremolinos (Málaga) (2008)

Nauck, D., Klawonn, F., Kruse, R.: Foundations of Neuro-Fuzzy Systems. John Wiley & Sons, Inc. (1997)

Nauck, D., Kruse, R.: A neuro-fuzzy method to learn fuzzy classification rules from data. Fuzzy Sets and Systems **89**(3), 277–288 (1997). DOI 10.1016/ S0165-0114(97)00009-2

Nauck, D., Kruse, R.: Neuro-fuzzy systems for function approximation. Fuzzy Sets and Systems **101**(2), 261–271 (1999). DOI 10.1016/S0165-0114(98) 00169-9

Novák, V., Perfilieva, I., Močkoř, J.: Mathematical Principles of Fuzzy Logic. Kluwer Academic Publishers, Dordrecht, Netherlands (1999)

Olaru, C., Wehenkel, L.: A complete fuzzy decision tree technique. Fuzzy Sets and Systems **138** (2), 221–254 (2003). DOI 10.1016/S0165-0114(03) 00089-7

Pavelka, J.: On fuzzy logic i, II, III. Zeitschrift für Mathematische Logik und Grundlagen der Mathematik **25**, 45–52, 119–134, 447–464 (1979)

Schröder, M., Petersen, R., Klawonn, F., Kruse, R.: Two paradigms of automotive fuzzy logic applications. In: M. Jamshidi, A. Titli, L. Zadeh, S. Boverie (eds.) Applications of Fuzzy Logic: Towards High Machine Intelligence Quotient Systems, *Environmental and Intelligent Manufacturing Systems Series*, vol. 9, pp. 153–174. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1997)

Smith, N.J.J.: Vagueness and Degrees of Truth. Oxford University Press, New York, NY, USA (2009)

Stevens, S.S.: On the theory of scales of measurement. Science **103**(2684), 677–680 (1946). DOI 10.1126/science.103.2684.677

Sudkamp, T.: Similarity, interpolation, and fuzzy rule construction. Fuzzy Sets and Systems **58**(1), 73–86 (1993). DOI 10.1016/0165-0114(93) 90323-A

Sugeno, M., Yasukawa, T.: A fuzzy-logic-based approach to qualitative modeling. IEEE Transactions on Fuzzy Systems **1**(1), 7–31 (1993). DOI 10. 1109/TFUZZ.1993.390281

Takagi, T., Sugeno, M.: Fuzzy identification of systems and its applications to modeling and control. IEEE Transactions on Systems, Man, and Cybernetics **15**(1), 116–132 (1985)

van Deemter, K.: Not Exactly: In Praise of Vagueness. Oxford University Press, New York, NY, USA (2010)

Wang, L., Mendel, J.M.: Generating fuzzy rules by learning from examples. IEEE Transactions on Systems, Man, and Cybernetics **22**(6), 1414–1427 (1992). DOI 10.1109/21.199466

Zadeh, L.A.: Fuzzy sets. Information and Control **8**(3), 338–353 (1965). DOI 10.1016/S0019-9958 (65)90241-X

Zadeh, L.A.: Probability measures of fuzzy events. Journal of Mathematical Analysis and Applications **23**(2), 421–427 (1968). DOI 10.1016/ 0022-247X(68)90078-4

Zadeh, L.A.: Outline of a new approach to the analysis of complex systems and decision processes. IEEE Transactions on Systems, Man, and Cybernetics **3**(1), 28–44 (1973)

Zadeh, L.A.: Fuzzy logic and approximate reasoning. Synthese **30**(3-4), 407–428 (1975). DOI 10.1007/BF00485052

Zadeh, L.A.: Fuzzy sets as a basis for a theory of possibility. Fuzzy Sets and Systems **1**(1), 3–28 (1978). DOI 10.1016/0165-0114(78)90029-5. (Reprinted in Fuzzy Sets and Systems 100 (Supplement 1), 9-34, (1999))

Zadeh, L.A.: A theory of approximate reasoning. In: J.E. Hayes, D. Michie, L.I. Mikulich (eds.) Proceedings of the Ninth Machine Intelligence Workshop, no. 9 in Machine Intelligence, pp. 149–194. John Wiley & Sons, Ltd., New York, NY, USA (1979)

Zadeh, L.A.: The role of fuzzy logic in the management of uncertainty in expert systems. Fuzzy Sets and Systems **11**(1–3), 197–198 (1983). DOI 10.1016/S0165-0114(83)80081-5

Zadeh, L.A.: Toward a perception-based theory of probabilistic reasoning with imprecise probabilities. Journal of Statistical Planning and Inference **105**(1), 233–264 (2002). DOI 10.1016/S0378-3758(01)00212-9

Zadeh, L.A.: Generalized theory of uncertainty (GTU)-principal concepts and ideas. Computational Statistics & Data Analysis **51**(1), 15–46 (2006). DOI 10.1016/j.csda.2006.04.029

Zadeh, L.: Toward human level machine intelligence - is it achievable? the need for a paradigm shift. IEEE Computational Intelligence Magazine **3**(3), 11–22 (2008). DOI 10.1109/MCI.2008.926583

Chapter 17
Statistics of the Field*

Frances Rosamond

In attempting to collect statistics on computer science, we are confronted (as were the authors of the COSERS book) with three problems. First, as remarked in that Introduction (Chap. 1), we face the lack of a precise definition of the field; today, due to the pervasiveness of computer-based research, computer science has become intertwined with other disciplines (e.g., an academic with a Ph.D. in computer science can be found in departments of education or biology). Second, due to the breadth of job scope, computer science data sometimes is grouped within Science and Engineering, and in other cases within Physical Sciences, or Mathematical Sciences. Third, specialization within the profession in terms of theory, software, as well as hardware development, results in rapidly changing statistics on computer science trends – government agencies, professional societies, corporations, books, blogs and wikipedias are prolific with a wealth of such information. Sorting through the multitude of data is the reverse problem of the sparse statistics that COSERS faced in the 1970s.

The primary computing associations include the Association for the Advancement of Artificial Intelligence (AAAI), Association for Computing Machinery (ACM), the Computing Research Association (CRA), the Institute of Electrical and Electronics Engineers-Computer Society (IEEE-CS), Society for Industrial and Applied Mathematics (SIAM), and Advanced Computing Systems Professional and Technical Association (USENIX). Each of these associations has an extensive website with focused information about their computing community throughout the world.

**Editor's preface* This chapter is a statistical summary of the state of Computer Science as of 2011. There are five sections with headings as follows: Education, Publishing, Funding, Employment, and Professional Associations. Figures are collected at the end of each section. Dr. Rosamond has performed a service to the Computer Science community.

F. Rosamond (✉)
School of Engineering and Information Technology,
Charles Darwin University, Casuarina, Australia
e-mail: frances.rosamond@CDU.edu.au

E.K. Blum and A.V. Aho (eds.), *Computer Science: The Hardware, Software and Heart of It*, DOI 10.1007/978-1-4614-1168-0_17,

The U.S. National Science Foundation and the Department of Education both collect a wide variety of science-related educational data, analyze trends (e.g., computer use, Ph.D. production), and contribute indicators to government and institutional policy. Pointers to their reports are available at the Integrated Sciences and Engineering Resources Data System (WebCASPAR) (https://webcaspar.nsf.gov/). Data on educational trends in science and research is also provided by the U.S. National Research Association and National Academies of Science (NRA/NAS), the Department of Labor (DoL), and the Bureau of the Census.

Under the National Science Foundation, the Division of Science Resources Statistics (SRS) provides a central clearinghouse for the collection and analysis of data on scientific and engineering resources, and provides information for policy formation by other Federal agencies. Several surveys are partially funded by other agencies including the National Center for Education Statistics, the Department of Energy, NASA, and the Bureau of the Census. SRS works collaboratively with international organizations such as OECD and UNESCO.

The NSF Division SRS produces the *Sciences and Engineering Indicators* (SEI) on the scope, quality and vitality of U.S. and international science and engineering activity. The SEI is policy neutral, does not model projections and avoids strong claims. Care is taken to present indicators in clear language using readily understandable analysis in order that the data are accessible to users with different needs and backgrounds. Indicators are subject to extensive review by outside experts, interested federal agencies, National Science Board members, and NSF internal reviewers for accuracy, coverage, and balance. The data are freely available online, together with tables, figures, links, and reference lists. Data from the *Science and Engineering Indicators 2010* have been used in this chapter (http://www.nsf.gov/statistics/).

The NSF Division SRS is also responsible for the Survey of Earned Doctorates (SED), which since 1957 has annually asked all individuals receiving U.S. research doctorates their field, institution, sex, and much more. In 2008, about 92% of the 48,802 new research doctorates completed the survey. The data have been collected annually since 1957 by six federal agencies: the National Science Foundation (NSF), National Institutes of Health (NIH), U.S. Department of Education (ED), U.S. Department of Agriculture (USDA), National Endowment for the Humanities (NEH), and National Aeronautics and Space Administration (NASA).

Under the U.S. Department of Education, the National Center for Education Statistics (NCES) is responsible for the Integrated Postsecondary Education Study Data System Survey (IPEDS), which provides a variety of data on the almost 10,000 public and private US postsecondary institutions. The NCES also conducts complementing studies of postsecondary faculty, degree recipients, financial aid, and transcript data, for example (http://nces.ed.gov/).

Since 1974, the Computing Research Association has conducted the annual Taulbee Survey to document trends in student enrollment, degree production, employment of graduates, and faculty salaries. The survey is sent to 264 PhD-granting departments in computer science (CS), computer engineering, and information technology in the United States and Canada (the Forsythe list).

The survey is named after Orrin E. Taulbee, who conducted these surveys from 1974 to 1984 for the Computer Science Board (the predecessor to the CRA). CRA's primary mission is to influence policy that impacts on computing research and development – information technology, cybersecurity, IT workforce, defense, and impediments to research. Reports are archived on CRA's website. An annual Computing Leadership Summit is conducted by CRA, and further evaluates the influence and recognition of computer science compared to other sciences.

Workforce information, such as computing people employed in other industries, can be found at www.acinet.org. The Information Technology & Innovation Foundation at www.itif.org also has a number of reports and articles that are of interest.

This chapter is divided into four main sections with the headings: – Education, Publishing, Funding and Employment. Data from the above and other sources identified in the references have informed this chapter, and these sources provide annual updates. A final fifth section on Associations provides a brief description of CS societies and agencies.

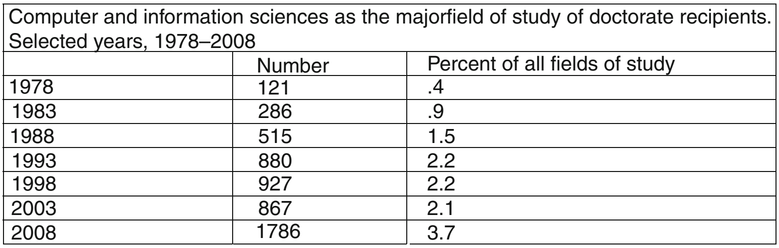
Statistics Part 1: Education

Production of Ph.D. Degrees in Computer Science

In 2008, the number of doctorate recipients claiming their major field of study as computer and information sciences reached a high of 1786, which is 3.7% of the total PhDs in all fields of study. The NSF data below shows that in 1978 there were 121 doctorate recipients, which was 0.4% of the total of PhDs (Fig. 17.1). There have been a total of about 22,000 computer science PhDs produced in the past 30 years.

The annual production of PhDs appears to roughly follow the US economic progress. According to the NCES, there were 248 PhDs in computer and information sciences awarded in 1984–1985. Over the following 10 years, doctorates increased to 887 in 1994. For the 8 years from 1994–1995 to 2001–2002, there was a steady decrease to 752. After 2002, numbers increased again, to 1698 in 2007–2008 (This is a difference of about 100 doctorates from the NSF data). The graph below, Fig. 17.2, was constructed using NCES values, and the numbers differ slightly from NSF or CRA data.

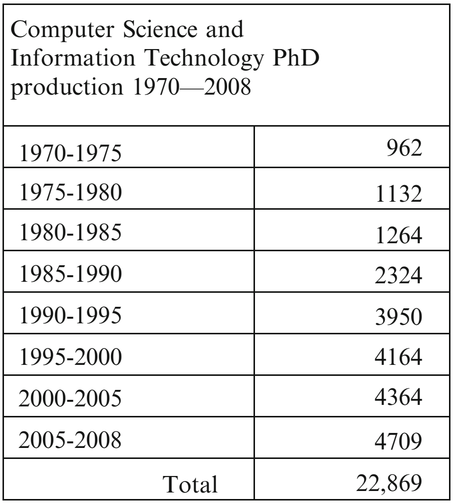
The 2008 Ph.D. production may have been a peak. The 2008–2009 CRA Taulbee Survey report that the numbers of "Computer Science" Ph.D.s have declined 7.8% from 2008. There were 147 out of 188 CS departments from the US and 41 out of 81 from Canada who responded to that Survey, about 71 percent. The decline was predicted based on declining numbers of new students in doctoral programs beginning in 2002–2003, which has been attributed to the "dot com" bust, increased immigration requirements on foreign students following September 11, and publicity on offshoring of computer jobs. Visa processing has since been streamlined. The number of new students entering CS doctoral programs in 2009



Computer and information sciences as the majorfield of study of doctorate recipients. Selected years, 1978–2008

	Number	Percent of all fields of study
1978	121	.4
1983	286	.9
1988	515	1.5
1993	880	2.2
1998	927	2.2
2003	867	2.1
2008	1786	3.7

Fig. 17.1 Major field of study of doctoral recipients
Source: National science foundation, division of science resources statistics. 2009. *Doctorate recipients from U.S. universities: summary report 2007–2008*. Table 5. Special report NSF 10–309. Arlington, VA. Available at http://www.nsf.gov/statistics/nsf10309/



Computer Science and Information Technology PhD production 1970—2008	
1970-1975	962
1975-1980	1132
1980-1985	1264
1985-1990	2324
1990-1995	3950
1995-2000	4164
2000-2005	4364
2005-2008	4709
Total	22,869

Fig. 17.2 PhDs in computer science and information technology 1980–2008
Source: U.S. Department of education, national center for education statistics, higher education general information survey (HEGIS), "degrees and other formal awards conferred" surveys, 1970–1971 through 1985–1986; and 1986–1987 through 2007–2008 integrated postsecondary education data system, "completions survey" (IPEDS-C:87–99), and Fall 2000 through Fall 2008 (This table was prepared July 2009)

is about the same as in 2008, although a larger percentage are from outside North America (see the section on foreign students below). The production of Master's degrees declined 6.7% in 2008–2009; however, new enrollments held steady.

According to the U.S. Department of Education data cited above, Bachelor's degrees in computer and information science reached a high of 42,337 in 1985–1986, and declined for the next 10 years to 24,506 in 1995–1996. The downturn ended and there began an increase to more than double (59,488) in 2003–2004. The 2010 UNESCO Report on the Sciences confirms that the number of CS Bachelor's degrees increased more sharply from 1998 to 2004 than any other science and engineering field except social science. However, this was followed by another 5-year downturn to 38,476 in 2007–2008. And, according to the 2008–2009 CRA Taulbee Survey, Bachelor's degree production in 2009 declined 12% from 2008. This will impact on lower numbers of future PhDs. However, the downturn may end in about 2012–2013. The 2008–2009 Taulbee Survey reports that numbers of new computer science majors has increased, up 5.5% from 2007 to 2008.

Women Earning Degrees in Computer Science

The share of doctorates earned by female U.S. citizens in 1985, 1995 and 2005 grew steadily and doubled in the physical sciences (from 16.5% to 30.6%) and engineering (from 9.6% to 19.8%), but U.S. women's share of doctorates in mathematics and computer sciences in 1985 (16.6%) remained at 16.6% in 1995, and then rose by only a third to 23.6% by 2005, echoing a similar rise in women's doctorates in behavioral (44.7–60.3%) and life sciences (34.8–52.8%).

(Source: National Science Board (2008) Science and Engineering Indicators http://www.unesco.org/science/psd/publications/usr10_usa.pdf Figure 9)

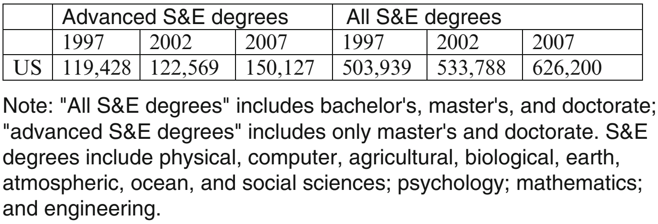
Between 2002 and 2008 the percentage of doctorates awarded to women in CS hovered at a little over 20%. In 2008–2009, women received slightly over 22% of Master's degrees in computer science, according to the 2008–2009 Taulbee Survey. However, that same survey showed women received only 11% of CS bachelor's degrees, and form only 18.4% of new PhD enrollment, indicating that more men may transition from undergraduate CS studies to jobs/other majors, while women may transition to jobs after earning Master's degrees.

The National Center for Women & Information Technology (NCWIT) produces an annual one-page "By the Numbers" report on women in technology which shows the contrast with 1985, when 37% of CS bachelor's recipients were women. According to NCWIT, there has been a 79% decline in the number of incoming undergraduate women interested in majoring in computer science between 2000 and 2008.

(Source: Women in Information Technology. By the Numbers 2008. (http://ncwit.org/pdf/BytheNumbers09.pdf))

Minorities Earning Degrees in Computer Science

The numbers of CS doctorates awarded to Blacks, Hispanics and American Indians and other minorities has been extremely small, each receiving less than 5% of degrees. None of these minorities exceeded 2% of North American PhD program enrollment in 2008 except for African-Americans (7.9%).



	Advanced S&E degrees			All S&E degrees		
	1997	2002	2007	1997	2002	2007
US	119,428	122,569	150,127	503,939	533,788	626,200

Note: "All S&E degrees" includes bachelor's, master's, and doctorate; "advanced S&E degrees" includes only master's and doctorate. S&E degrees include physical, computer, agricultural, biological, earth, atmospheric, ocean, and social sciences; psychology; mathematics; and engineering.

Fig. 17.3 Advanced S&E degrees as share of S&E degrees conferred 1997, 2002, and 2007
Source: portion of Table 8.20 from national center for education statistics, integrated postsecondary education data system (various years). Science and engineering indicators 2010

(Source: 2007–2008 Taulbee Survey, Table 8, "*PhD Program Total Enrollment by Ethnicity*," Computing Research News, Vol. 21/May 2009.)

State-by-state NCES data on a wide variety of educational issues, including employment and productivity is available on the website. Often, computer science is grouped into Science and Engineering (S&E), which includes physical, computer, agricultural, biological, earth, atmospheric, ocean, and social sciences; psychology; mathematics; and engineering. Taken as a group (see Fig. 17.3), about 24% achieve advanced S&E degrees (master's and doctorate) when compared to all S&E degrees (advanced plus bachelor's), although this varies by state.

According to the 2008 SED the median number of 7 years to receive a doctorate in the physical sciences (which includes mathematics and computer and information sciences) since starting graduate school, has remained constant since 1983, varying slightly between types of institutions. At research institutions with very high research activity, the median number of years was less than at other universities (possibly indicating that students at less prestigious universities need outside jobs while studying). The majority of doctorate recipients in S&E fields earn their degrees while in their early 30s; this may be a factor of gender roles allowing men to focus on their careers. Doctoral recipients' average age is nearly 35 years in humanities and 42 years in education, recalling that women (who comprise most minority students in these fields) may return to universities after raising families and experiencing relevant social issues.

(Source: Table 18 Field of study and time to degree. Selected years 1983–2008. Table 20 Age at doctorate. NSF/NIH/USED/USDA/NEH/NASA, 2008 Survey of Earned Doctorates (SED). http://www.nsf.gov/statistics/nsf10309/content.cfm?pub_id=3996&id=5)

In 2007–2008, the average total price (tuition and fees, books and materials, and living expenses) for 1 year of full-time graduate education was $37,300 for a master's degree program and $42,800 for a doctoral program (in 2008–2009 dollars), differing by type of institution (public or private). About 50% of full time college students ages 16–24 were employed, with about 10% working 35 or more hours per week. About 80% of part-time college students were employed. Only 26% of master's degree students were enrolled full time in 2007–2008, compared to 53% of doctoral degree students. About 85% of full-time students at the master's level and 93% at the doctoral level received some type of aid. (Grants and assistantships are usually not related to financial need. Financial need must be demonstrated by students in order to obtain Perkins or subsidized Stafford awards.)

(Source: U.S. Department of Education, National Center for Education Statistics, 2007–2008 Integrated Postsecondary Education Data System, IPEDS, Spring 2009. *Section 5 – Contexts of Postsecondary Education*, p. 137.)

President Obama is bringing attention to the fact that the U.S has fallen from 1st to 12th place in college graduation rates in a single generation, per the August 2010 U.S. College Board "College Completion Agenda" (http://completionagenda.collegeboard.org); he wants to raise U.S. college graduation rates to 60% in just 10 years, adding at eight million college graduates. In 2007, 40.4% of U.S. 25- to 34-year-olds held degrees, far short of 55 + % for Canada, Korea and Russia.

Foreign Students Earning Degrees in Computer Science

Non-U.S. students are much more likely to enroll in computer science and engineering at all levels than U.S. students, and this has caused some concern. According to the 2006 SED, non-U.S. citizens accounted for 65% of doctorates in computer science, and the increase in S&E doctoral awards to non-U.S. citizens was three times larger than to U.S. citizens. As indicated in Fig. 17.4, since 1994, over half of all doctorates earned in mathematics and engineering and close to half of those in computer science have been foreign students.

The NSF table (Fig. 17.5) shows that by 2008, numbers of temporary visa holders almost equal those of U.S. citizen doctorate recipients in the physical sciences, which includes computer science, and has exceeded U.S. citizens in engineering. As mentioned above, the 2008–2009 Taulbee Survey reports that the number of new computer science and information technology doctoral students from outside North America rose from 54% in 2008 to 59% in 2009.

Students from five countries make up the majority of all foreign doctoral students: the People's Republic of China (PRC), South Korea, Taiwan, India, and Canada. Between 1985 and 2005, students from China, India and the Republic of Korea earned half or more of all doctorates in S&T fields of computer sciences, mathematics, physics and engineering awarded in the USA to students from foreign countries (Fig. 17.6). Students from these and other countries come very well trained in both science and study skills, and provide strong competition with U.S. students, and after graduation in the job market. Foreign doctorate recipients

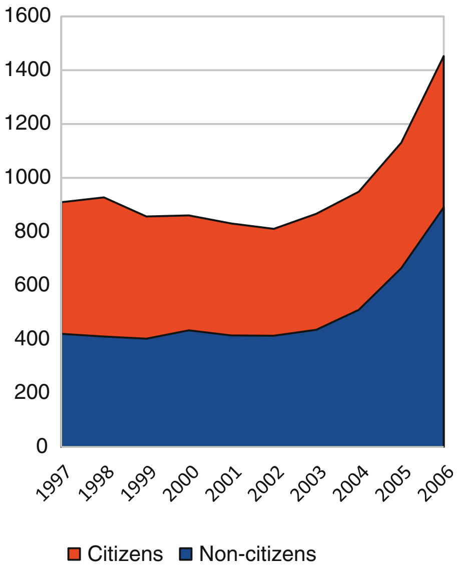
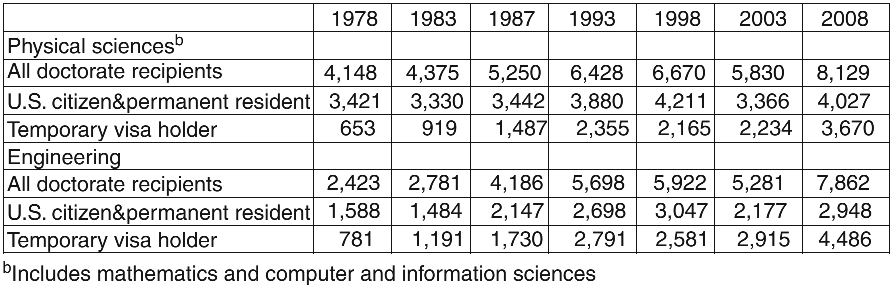


Fig. 17.4 CS doctoral degrees: US citizens and non-citizens
Source: National science foundation, division of science resources statistics. 2009. *Doctorate recipients from U.S. Universities: summary report 2007–2008*. Special report NSF 10–309. Arlington, VA. (http://www.nsf.gov/statistics/nsf10309/)



	1978	1983	1987	1993	1998	2003	2008
Physical sciences[b]							
All doctorate recipients	4,148	4,375	5,250	6,428	6,670	5,830	8,129
U.S. citizen&permanent resident	3,421	3,330	3,442	3,880	4,211	3,366	4,027
Temporary visa holder	653	919	1,487	2,355	2,165	2,234	3,670
Engineering							
All doctorate recipients	2,423	2,781	4,186	5,698	5,922	5,281	7,862
U.S. citizen&permanent resident	1,588	1,484	2,147	2,698	3,047	2,177	2,948
Temporary visa holder	781	1,191	1,730	2,791	2,581	2,915	4,486

[b] Includes mathematics and computer and information sciences

Fig. 17.5 Citizenship status of doctorate recipients, by broad field of study: selected years, 1978–2008
Source: Portion of Table 11 from NSF/NIH/USED/USDA/NEH/NASA, 2008 survey of earned doctorates. (http://www.nsf.gov/statistics/nsf10309/content.cfm?pub_id=3996&id=8#tab5)

include permanent and temporary residents who have tended to remain in the United States to work, resulting in significant numbers of foreign university faculty in the scientific disciplines (see Fig. 17.7) and foreign doctorates employed by industry.

Countries such as China, India and South Korea that traditionally have sent students to the US to study, are building their own new universities (sometimes as

	Computer Science	All S&E
China	2,166	50,220
India	1,791	21,354
South Korea	849	20,549
Taiwan	959	18,523

Fig. 17.6 Foreign recipients of US doctorates: 1987–2007
Source: Portion of Table 2.5: foreign recipients of US S&E doctorates, by country/economy of origin: 1987–2007 and Table 2.6: foreign recipients of US CS doctorates, by country/economy of origin: 1987–2007 from national science foundation, division of science resources statistics, survey of earned doctorates, special tabulations (2009)

Faculty with Non-resident Alien Status				
	2005-2006		2008-2009	
	N	%	N	%
Full	3	0.2	6	0.3
Associate	19	1.6	35	2.6
Assistant	178	15.7	147	16.6
Teaching	10	1.5	16	2.5
Research	44	11.4	77	16.3
Post-Docs	83	31.8	165	37.5
TotaL	337	6.3	446	8

Fig. 17.7 Foreign academics in US universities
Source: Portion of Table 22: ethnicity of current faculty, non-resident Alien. CRA 2005–2006 Taulbee survey (www.cra.org)

big as small cities) and competing for the best students. As these countries increasingly become technology leaders, foreign students who previously would have wanted to remain in US after graduation, may more likely return home to work in a local university or research lab. This has led to concerns that the U.S. may face a lack of computer science researchers and IT professionals in the future.

On the other hand, according to a Brookings Institute Report by Ben Wildavsky (Academic Globalization Should Be Welcomed, Not Feared, in the New York Academy of Sciences Magazine, October 22, 2010. http://www.brookings.edu/articles/2010/), Tsinghua and Peking universities together recently surpassed Berkeley as the top sources of students who come to American to earn PhDs. As previously stated, foreign students dominate doctoral programs, constituting over 65% of Ph.D.s in computer science.

Public impression seems to be that large numbers of foreign students choose to remain in the US after graduation, and become employed in US universities. The CRA Taulbee Survey provides data on numbers of non-resident aliens employed at all academic levels, as well as numbers of other ethnicity groups, and positions filled and unfilled by department rank and by academic rank.

The US encourages international collaboration in many ways. For example, the third U.S.-China Computer Science Leadership Summit, jointly sponsored by the National Natural Science Foundation of China (NSFC) and the U.S. NSF was held in Peking University during June, 2010. Altogether, 18 deans of the schools of computer sciences from among the top 30 U.S. universities in the field of computer sciences, 39 deans of the schools of computer sciences from among China's top 30 universities and institutes in the same field, as well as some senior researchers attended the summit.

Curriculum

Computer science research inter- and across disciplines has grown so significantly and diversely that the National Academy of Science (NAS) and the National Academies Board on Higher Education and Workforce convened a Committee on Taxonomy to review the classification of CS fields found in the Doctorate Records File (DRF), which is maintained by the NSF (as lead agency for a consortium that includes the NIH, USDA, NEH, and ED). Computer Sciences is a category listed under the Physical Sciences and Mathematics.

1. Physical Sciences and Mathematics
   (a) *Computer Sciences*
      - Artificial Intelligence/Robotics
      - Computer and Systems Architecture
      - Databases/Information Systems
      - Graphics/Human Computer Interfaces
      - Numerical Analysis/Scientific Computing
      - Programming Languages/Compilers
      - OS/Networks
      - Software Engineering
      - Theory/Algorithms

The Taxonomy also includes a category of *Emerging Fields* (not quite enough graduates or doctoral curricula to be considered a ¨field¨) which include Computational Engineering and Information Science, Bioinformatics and Biotechnology, and Computational Linguistics.

(Source: National Academy of Sciences Board on Higher Education and Workforce. All rights reserved. 500 Fifth St. N.W., Washington, D.C. 20001. http://sites.nationalacademies.org/PGA/Resdoc/PGA_044522, Revised 7/31/06)

The broad sweep of computer science courses within a university was well described by Jeannette Wing, then Head of the Computer Science Department at Carnegie Mellon, in a 2005 presentation to the National Center for Women and Information Technology (NCWIT). Wing reports that the CS Department at Carnegie Mellon is the home for traditional areas of computer science, but also home for conjunction with other disciplines, such as:

- Robotics: CS + Mechanical Engineering + Electrical Engineering
- Language Technologies: CS + Linguistics
- Human-Computer Interaction: CS + Design + Psychology
- Automated Learning and Discovery: CS + Statistics
- Software: CS + Public Policy + Management
- Entertainment: CS + Drama

Multidisciplinary degrees featuring computer science range from the MIT *Leaders for Global Operations* program, which is joint between the School of Engineering and the Sloan School of Management; the Harvard Center for Research on Computation and Society, which includes neuroscience; and Cornell's Computational Synthesis Lab which explores biological concepts in engineering design. Programs featuring computer science flourish outside of CS departments as well; for example, the Cornell Institute for Computer Policy and Law (EDUCAUSE), and the New Medium Consortium (NMC).

So what is computer science? Since the 1960s, ACM and the Computer Science Teacher Association (CSTA) along with other leading professional and scientific computing societies, have provided curriculum standards. In 2001, a five-volume series of Curriculum Guidelines on Computer Engineering, Computer Science, Information Systems, Information Technology and Software Engineering together with an Overview volume for undergraduate and graduate degree programs was published. In 2010, a second edition of Model Curriculum for K-12 Computer Science was provided. The Computer Science curriculum was updated in 2008 (from the CS2001 Body of Knowledge). The review has 108 pages, with course descriptions included in order to help departments implement the recommendations. Extensive industry involvement was solicited in an attempt to respond to the crises of low enrollments (a plummet of as much as 60–70% from the peak of 2001).

Key recommendations for curriculum in 2008 included an updating of all topics from 2001, emphasizing concurrency, net-centric computing, human computer interaction, software engineering, management information systems, systems issues and professional practice. A second key recommendation was to address issues of security systematically, both in programming, as well as in operating systems and networking. Trends in student theses showed these topics to have become increasingly relevant. In section "Statistics Part 2: Publishing" of this chapter, we also see the relevancy of these new CS curriculum additions reflected in new CS job specialties; the Taulbee Survey on PhD Employment by Specialty has added security, networks, robotics/vision, bioinformatics, information science and systems, and social informatics as CS job specialties since 2007.

CS curriculum updating has led to increasing discussions about what constitutes computational thinking, and a number of perspectives are presented in Report of a Workshop on the Scope and Nature of Computational Thinking, published by the National Academy of Sciences, 2010.

Increasingly sophisticated measures are being used to assess the quality and effectiveness of doctoral programs. For example, for the 2005–2006 academic year, the NRC/National Academies Press reviewed more than 5,000 doctoral programs at

212 universities with data covering faculty publications, grants, and awards, program size, and many other characteristics, as well as ranges of rankings for research activity and other dimensions of program quality. The massive dataset: *Data-Based Assessment of Research-Doctorate Programs in the United States*, is available from the National Academies Press (2007) (http://www.nap.edu/rdp/). The program rankings reported by US News and World Reports receive a lot of attention.

The internet has provided an alternate access to CS education through the large number of distance learning university programs. Three of the top ten institutions awarding degrees in computer science in 2007 were online, with the University of Phoenix Online Campus offering about 2000 computer science bachelor's degrees, more than double that of any other university.

The internet also provides many free online video lectures and coursework at projects like the Massachusetts Institute of Technology OpenCourseWare, AcademicEarth.org, and iTunes-U.

Additional initiatives are being made to accelerate the development and use of online learning tools. In 2010, The Bill and Melinda Gates Foundation, the William and Flora Hewlett Foundation, and four nonprofit education organizations (Educause, the Council of Chief State School Officers, League for Innovation in the Community College and International Association for K-12 Online Learning) are creating an initiative featuring online learning with the goal of increasing educational opportunities, especially for low-income young adults.

(Source: The New York Times, Business Day Technology online, In Higher Education, a Focus on Technology by Steve Lohr, Published: October 10, 2010.)

Science, Technology, Engineering, and Mathematics (STEM)

Maintaining technological superiority in the U.S. depends on enlarging the pipeline of future CS PhD recipients, part of what is labeled as "STEM" production. According to international comparisons compiled by NSF, the U.S. has one of the lowest rates of STEM to non-STEM degree production in the world. In 2002, STEM degrees accounted for only 16.8% of all first university degrees awarded in the United States, while the international average was 26.4%. Students in the U.S. prefer to study business and psychology, rather than mathematics.

(Sources: National Science Foundation, Science and Engineering Indicators, 2006, Volume 1, Arlington, VA, NSB 06–01, January 2006, Table 2.37. Also, international data on academic postsecondary programs (ISCED levels 5A and 6) in 2004 corresponding to bachelor's, master's, first-professional, and doctoral degrees in the US, collected through the Organization for Economic Cooperation and Development (OECD), (http://nces.ed.gov/programs/coe/2007/section5/indicator43.asp))

According to the US Department of Education, our students lack interest and ability in mathematics and science. The US ranks 25th of 30 OECD countries in math literacy. One quarter of U.S. fifteen-year-olds do not reach the baseline level of science or mathematics competence.

(Source: Steve Robinson, U.S. Department of Education, White House Domestic Policy Council February 22, 2010.)

Low levels of U.S. student interest and scores in math and science are of great national concern, and there have been many efforts to improve the ways US students learn science, mathematics, technology and engineering (e.g., the National Innovative Initiative, C-PATH by NSF; Building Engineering & Science Talent [BEST]; The Merck Institute for Science Education; Project Kaleidoscope; and many others). A long list of initiative reports can be found on the STEM Coalition website (http://nstacommunities.org/stemedcoalition/reports/).

Attention to STEM is promoted by a wide variety of associations and agencies. For example, the STEM Education Coalition is composed of advocates from over 1,000 diverse groups representing all sectors of the technological – for example, computer science – workforce: knowledge workers, educators, scientists, engineers, and technicians. The Coalition is co-chaired by the American Chemical Society and the National Science Teachers Association, and works aggressively to raise awareness in Congress, the Administration, and other organizations about the critical role that STEM education plays in enabling the U.S. to remain competitive in the twenty-first century. Widely dispersed efforts can be a difficulty, according to CRA, since organizations and government agencies (such as the NSF, NIST, NASA, the Census Department, and the National Labs) all compete for government funding for STEM projects.

In 2005, the National Academy of Sciences published “Gathering Storm”, a study calling for investment in science, technology and education. Their top three recommendations were to:

1. Increase America’s talent pool by improving K-12 science and mathematics education;
2. Strengthen teachers’ skills through additional training in science, math and technology; and
3. Increase the pool of students prepared to enter college and graduate with STEM degrees.

In 2010, an updated version: “Rising Above the Gathering Storm, Revisited: Rapidly Approaching Category Five”, reported on the severely compromised status of U.S. student achievement in science, in spite of government and private sector efforts; “The outlook for America to compete for quality jobs has further deteriorated over the past five years”. Among reinforcement initiatives, the NAS proposed the creation of 5,000 new fellowships each year, with an annual stipend of $30,000–$50,000 for doctoral degrees, to be administered by the NSF.

The NSF has numerous programs in STEM education, from primary through graduate school. In early 2009, the economic downturn caused universities and companies to severely curtail their hiring of new PhDs in computing fields. When it became clear that many new PhDs were in danger of falling out of research and education careers, the NSF supported the Computing Innovation Fellows (CIFellows) Program (through the CCC of CRA), to create opportunities for at least some new PhDs to start careers at top research and education organizations, thereby saving the large investments that have been made in their training and education.

The Graduate Research Fellowships is the largest of the NSF STEM education programs, and represents one of the longest-running federal STEM programs (enacted in 1952). The program provides three years of support to approximately 1,000 graduate students annually in STEM disciplines who are pursuing research-based master's and doctoral degrees, with additional focus on women in engineering and computer and information sciences. In 2006, there were 907 awards given to graduate students studying in nine major fields at 150 institutions. The Research Experiences for Undergraduates (REU) program is the largest of the NSF STEM education programs that supports active research participation by undergraduate students, and in 2010 focused on funding centers for cyberinformatics Bachelor's degree studies at Louisiana and Oregon State Universities.

The NSF Broadening Participation in Computing (BPC) program funds eleven alliances involving a diverse set of institutions – large research universities, historically black colleges, states, middle and high schools, and various non-profit organizations. The goal is to leverage their faculty and financial resources to encourage more students to pursue computer science degrees, with special emphasis on underrepresented minorities. First distributed in 2005, funding grants are for three years, with the potential for an additional two-year extension. An evaluation report by the AAAS: Telling the Stories of the BPC Alliances: How One NSF Program Is Changing the Face of Computing, (Nov 2010) states that while the number of students pursuing computer science degrees has declined nationally, great success has been seen in institutions participating in the BPC.

In the State of the Union Address in 2006, President George W. Bush announced the American Competitiveness Initiative (ACI), called the "America Competes Act". Bush proposed the initiative to address shortfalls in federal government support of educational development and progress at all academic levels in the STEM fields. The initiative called for significant increases in federal funding for advanced R&D programs, including a doubling of federal funding support for advanced research in the physical sciences through the U.S. Department of Energy. While political divisions temporarily halted ACI refunding, in October 2010 President Obama re-energized the initiative to gain STEM students and graduates by calling for the recruitment of 10,000 new STEM K-16 teachers (outcome 2) to train at federally funded R&D centers.

The National Aeronautics and Space Administration (NASA) funding is not included in ACI, but has programs and curricula to advance STEM education at all levels. For example, in the *NASA Means Business* competition, sponsored by the Texas Space Grant Consortium, college students compete to develop promotional plans to encourage students in middle and high school to study STEM subjects, and to inspire professors in STEM fields to involve their students in outreach activities that support STEM education.

The Science and Mathematics Access to Retain Talent (SMART) Grants provide up to $4,000 for each of the third and fourth years of undergraduate study and are in addition to the student's Pell Grant award (http://www.fas.org/sgp/crs/misc/RL33434.pdf).

Other programs to encourage student interest include contests, such as the American Computer Science League and Computer Olympiad contests for secondary students, and SAT contests and SONY Robot for university students. There are summer schools and mentoring programs by ACM, IEEE, and others. There are various awards for computing expertise, such as the million dollar prize awarded by NetFlix in 2009 for improvements to their movie recommendation process, www.DuPont.com/Science_Awards, Economist's Innovation Awards, ideashappen.msn.com, www.imagencup.com, and many others. Perhaps the most well-known prize for professional theorists is the million dollar Clay Award for solving P versus NP.

Statistics Part 2: Publishing

The quality and number of refereed publications in prestigious journals continues to be a defining measure for individual academic researchers. However, the internet has changed the way we share, conduct and archive research. New ways of working together electronically, such as Polymath, allow for easy collaboration at a distance and on-line journals allow for dynamic updating of surveys. The most recent papers and slides are available on conference and personal websites. Traditional refereed journals often use electronic dissemination to cut costs and speed delivery of scientific and scholarly knowledge. The arXiv site allows researchers to maintain authorship and post results prior to publication. Publications are indexed and archived in huge digital libraries maintained by universities, government, professional organizations, publishers and search engines. There are numerous freely available blogs and online journals, such as the Electronic Journal of Combinatorics, Theory of Computing, and Chicago Journal of Theoretical Computer Science.

Productivity

In the U.S., the bulk of academic research and publications come from just 127 research universities that each obtain more than $15 M annually in federal grants. In 2010, America still leads the world in S&E publications, yet its share has slipped slightly from 2002 (31%). In 2008, the US produced about 168,000 scientific articles in S&E, almost 30% of the world total, but less than its 35% share in 1995. China produced about 60,000 articles (7.5% of total) but up 17% from 1995.

(Source: The Economist, Global science section, "Climbing Mount Publishable; The old scientific powers are starting to lose their grip" (Nov 11, 2010))

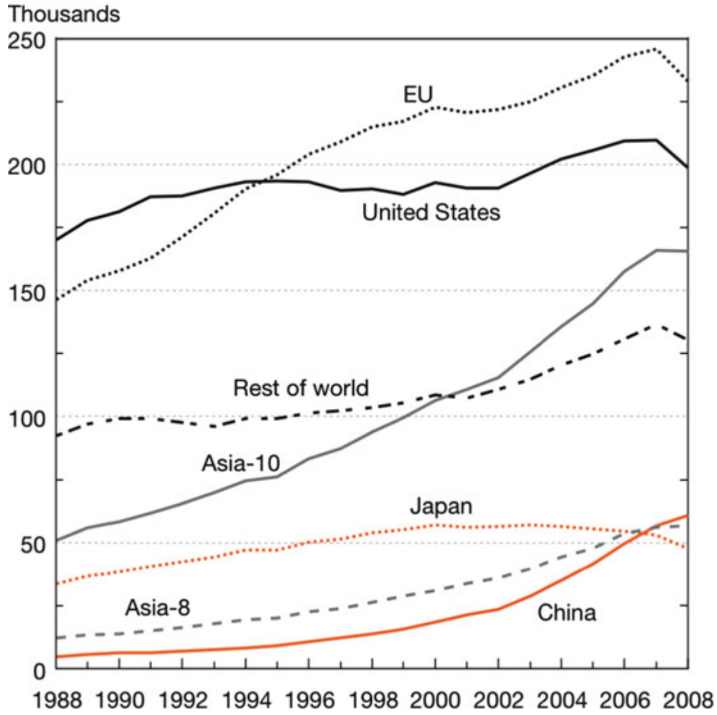


Fig. 17.8 Worldwide S&E journal article production. *EU* European union. Asia-8 includes India, Indonesia, Malaysia, Philippines, Singapore, South Korea, Taiwan, and Thailand. Asia-10 includes Asia-8 plus China and Japan. Internationally coauthored articles credited fractionally to authors' countries/locations. Counts for 2008 are incomplete

Source: Fig. O-13. Powerpoint Slides for S&E: Journal articles produced by selected regions/ countries: 1988–2008. NSF Science and Engineering Indicators 2008

The US has roughly 38% of the world citations. The U.S., U.K., EU and Japan exceeded/averaged 10 citations per S&T article published between 1998–2009, while China trailed South Korea in averaging five or less citations (Source: NBS and MOST (2010) China Statistical Yearbook on Science and Technology 2009; OECD (2009) Main Science and Technology Indicators, Volume 2009/1).

US production has remained roughly constant at about half-an-article annually per S&E doctorate holder in academia (indicating that the ratio of S&E doctorate holders in China is rapidly increasing by comparison). State-by-state S&E article output is reported in the NSF Science and Engineering Indicators.

The EU produced about 37% of world S&E articles in 2008, and had about 33% of world citations. In 2010, the EU's collective share has also fallen from 35% in 1995, whereas China's has reached 10%. In 2008, China produced about 60,000 articles (7.5% of total), triple its share from 1995. South Korea and Brazil's shares grew to 2.7% of the world's S&E publications, up by 60% from 1.7% in 2002, and less than 1% in 1995.

The data for the preceding paragraphs comes from article counts taken from Thomson Reuters, SCI and SSCI, http://thomsonreuters.com/products_services/science/; The Patent Board^{TM}; and National Science Foundation, Division of Science Resources Statistics. Appendix Table 5.25 and Table 8.42. Science and Engineering Indicators 2010.

Coauthorship

Scientists find it easier to work together across national borders, thanks to the internet. More than 35% of articles in leading S&E journals in 2010 are the result of international collaboration, up from 25% in 1995.

(Source: UNESCO Science Report 2010, quoting Chair, Royal Society)

In 2008, US S&E authors were most likely to coauthor with colleagues in UK, Germany, Canada or China, with those countries holding roughly 14, 13, 12, and 11% of respective shares of US international coauthored articles.

Computer science articles generally have far fewer co-authors than other fields. Computer science had about 1.9 co-authors in 1988, 2.6 in 2003, and 3 in 2008. The number of co-authors in all S&E fields increased from about 3 in 1988 to 4.7 in 2008, with the largest in medical science (5.6 in 2008).

(Source: Thomson Reuters, Science Citation Index and Sciences Citation Index, http://thomsonreuters.com/products_services/science/; The Patent Board^{TM}; and National Science Foundation, Division of Science Resources Statistics, special tabulations. Appendix Table 5.19, Table 5.16 and Table 5.22. Science and Engineering Indicators 2010)

The cost of scientific articles has been calculated by the NSF Division of Science Resources Statistics, Academic Research and Development Expenditures. They reported US$51,784 M million dollars in research and development expenditures in S&E for 2008, which resulted in about 3.24 articles per million dollars (down from 5.36 articles/million dollars in 1998), or about $180,000 per article. It is not clear how these dollars relate explicitly to computer science research, however, since S&E includes many disciplines including physics, and development may also include resources such as equipment or floor space.

(Source: Table 4.3 Science and Engineering Indictors 2010.)

The size of Asia's population leads UNESCO to conclude that it will become the "dominant scientific continent in the coming years". China is on the verge of overtaking both America and the EU in the quantity of its scientists. Each had roughly 1.5 m researchers out of a global total of 7.2 m in 2007. Nevertheless, in 2007 the number of scientists per million people remains relatively low (1,0181/m) in China versus 4,662/m in the U.S. and 4,181/m in the U.K. China has increased its global share of CS publications from 4.54% (1999–2003) to 10.66% by 2008, (Source: Adams et al. (2009) Global Research Report China: Research and Collaboration in the new Geography of Science and also planned in 2008 to double its 2008 IT funding budget, from $9.2B to $20.5B, an indication of its S&T priorities. India, already the world's leading exporter of information-technology services, and second only to China in the size of its population, has only a tenth as many researchers. This will change as both China and India improve educational access for their citizens, along with additional investment in IT infrastructure and CS curriculae.)

Peer Review

The cost of journal articles does not include a cost for the peer review process, which generally requires three reviewers to examine each article submitted to a conference or journal, and then final selection by a program committee or editor. The reviewers mostly are volunteers familiar with the subfield specialty topic. Electronic systems such as EasyChair help streamline the process, however it is still quite a burden on reviewers.

Over 25,000 CS articles annually are published or disseminated online since 1998, compared to 2,700 in 1975. There are over 132 major CS journals, and over 1,800 academic journals covering CS research in 2010, as compared with 28 in 1980. The J of ACM Hypertext Bibliography Project at MIT notes that the JACM Computing Reviews, begun in 1964, originally covered publications in 8 categories and 172 topics, a system (1964–1981) updated (1982 and 1991) to an 11-category, 4-level topic tree system now used. In 1998, the ACM Chief Editor, Dr. J. Halpern of Cornell University, also established the non-reviewed online Computing Research Repository (CoRR) to augment rapid dissemination of CS research papers, including conference proceedings and theses/dissertations, in 33 categories; foreign language authors are invited to participate in order to enable global use. In 1975, ACM reviewers covered no foreign-language articles, and few theses or conferences.

Impact Factor and H Factor

In addition to the type of articles written, the number of articles written, and how often each article is cited, academic evaluation includes the ranking of the journals in which the article is published. Journals are ranked in various ways by different countries or universities. In addition, the journal *impact factor* is used as a measure of the prestige of journals in which individuals have been published. The Thomson Reuters Impact Factor annual JCR impact factor is a ratio between citations and recent citable items published; e.g., the impact factor of a journal is calculated by dividing the number of current year citations to the source items published in that journal during the previous two years. For example:

A = total cites in 2012
B = 2012 cites to articles published in 2010–2011 (this is a subset of A)
C = number of articles published in 2010–2011
D = B/C = 2012 impact factor

Thomson Reuters began to publish Journal Citation Reports® (JCR®) in 1975 as part of the SCI and the Social Sciences Citation Index® (SSCI®). The JCR summarizes citations from more than 10,000 journals and proceedings in the sciences and social sciences indexed in the Web of Science database. Nearly 8,000 journals appear in the 2007 JCR, with detailed reports of their citation performance, their citation network, and the count and type of materials published.

In addition to the JCR Impact Factor, the JCR® includes the Eigenfactor™ Metrics, which use citing journal data from the entire JCR file to reflect the prestige and citation influence of journals by considering scholarly literature as a network of journal-to-journal relationships.

(Source: http://thomsonreuters.com/products_services/science/free/essays/impact_factor/) The Thomson Reuters Impact Factor by Dr. Eugene Garfield. Founder and Chairman Emeritus, ISI. Essay was originally published in the Current Contents print editions June 20, 1994, when Thomson Reuters was known as The Institute for Scientific Information® (ISI®).

Another index used to measure the productivity and impact of an individual scientist is the H-factor (after physicist Jorge E. Hirsch). The index is based on the set of the scientist's most cited papers and the number of citations that they have received in other people's publications. A scholar with an index of *h* has published *h* papers each of which has been cited by others at least *h* times. Individuals can compute their *h*-index manually using citation databases. Google Scholar or subscription-based databases such as Scopus and the Web of Knowledge provide automated calculators. Each database is likely to produce a different *h* for the same scholar, because of different journal coverage. The *h*-factor varies by discipline. In computer science, conference preprints are often excluded from the index. Conferences are important in computer science, but in most other fields are accorded less weight in evaluating academic productivity.

Conference citation gaps are not the only problem with the current CS conference situation. According to the Peer Review Panel at the 2010 CRA Snowbird Conference, CS subfields feel their research is not well represented and have splintered off into specialty conferences, possibly indicated by decreased attendance at STOC 2010 (350 participants compared to 500 in 1987). Authors are often the only attendees at conferences, since presentation of a paper generally is the only way to obtain travel funding. There is the possibility of missing big ideas and losing relevance, and becoming locked in a cycle of "deadline-driven" research.

Since even journals are limiting in terms of article time to publication and reach subscribers, the CS community has turned to free and subscription, reviewed and non-reviewed online blogs, archives, libraries, and bibliographies, to quickly capture all the latest and greatest CS ideas.

Blogs

Life in the computer science field is exhibited in online blogs. There are over 50 computer science blogs, from blogs for beginners to complex theoretical blogs that can challenge the best minds of the field (http://www.mastersincomputerscience.net/top-50-computer-science-blogs.html, Published by Madison on Wed May 5, 2010, accessed on 4 October 2010). According to mastersincomputerscience.net, there are 23 blogs on Computer Science, 20 on Computational Complexity and Theory, and seven on the juncture of Physics and Computer Science.

Digital Archives, Libraries, and Bibiliographies

Possibly one of the most significant of the electronic events is the arXiv. Researchers have had to be somewhat circumspect about publically discussing their results before the results are published – and publication traditionally has been a very long process, sometimes years.

The arXiv has changed all this by (almost) guaranteeing author ownership during the sometimes long gap between finishing the research and journal publication. Scientists upload their papers to arXiv.org for worldwide access. Results are disseminated in the fastest possible way. The arXiv was started in 1991 by Paul Ginsparg originally as a repository for preprints in physics and later expanded to computer science and other areas. It was originally hosted at the Los Alamos National Laboratory (hence its former address at `xxx.lanl.gov`) and is now funded by Cornell University and NSF, with mirrors around the world. The name and address was changed to `arXiv.org` in 1999 for greater flexibility. ArXiv versus snarXiv is a popular game of guessing which title refers to a genuine scientific article.

One of the quickest ways to find the publication record of an individual researcher is to use the Computer Science Digital Bibliography (DBLP) maintained by Michael Ley (Univ Trier http://dblp.uni-trier.de/), a tremendous index of bibliographic information on more than one million articles and containing more than 10,000 links to home pages of computer scientists. Access is free.

Archiving and electronic access to computer science material is provided by government, professional organizations, universities, companies, and national labs. Search engines such as Google Scholar locate millions of articles on webpages, while thousands of individuals build and update Wikipedia.

The National Science Digital Library (http://nsdl.org/) is the Nation's online library for education and research in Science, Technology, Engineering, and Mathematics. The NSDL hosts the Computer Science and Information Technology Gateways and Resources collection. The collection is comprised of web portals, sites, and individual digital resources devoted to research in computer science and information technology, as well as materials for the general public, and include resources in many areas such as algorithms and data structures, operating systems and programming languages, software engineering, artificial intelligence, information science, digital-library technologies, and others.

The National Labs offer access to their technical publications, such as at NASA Technical Reports Server (http://ntrs.nasa.gov/search.jsp). Companies also offer access. For example, CyberDigest: IBM Research Reports, offers the scientific community access to technical reports written by members of the IBM Research community (online at http://domino.research.ibm.com/comm/research.nsf/pages/d.compsci.html).

Two large digital libraries are maintained by the ACM and by the IEEE. The ACM Digital Library is a collection of citations and the full text of every article ever published by ACM, including journals, magazines, transactions, special interest group (SIG) newsletters, proceedings, and publications by affiliated organizations. ACM Computing Reviews (ISSN 1530–6586) is an academic journal that has reviewed computer science literature since 1964, and was an important resource for the original COSERS overview of CS Publication.

The ACM Computing Classification System (CCS) serves as one of the most generally used systems for the classification and indexing of the published literature of computing, and is the basis for classifying documents in the *ACM Guide to Computing Literature*. The *Encyclopedia of Computer Science* has a closely related taxonomy. The IEEE Computer Society Digital Library provides access to almost 310,000 articles and papers from 26 IEEE Computer Society periodicals and 3300+ conference publications. The IEEE Xplore is a subscription research database that mainly covers material from IEEE and IET, and contains over two million records.

CiteseerX (previously Citeseer) is a scientific literature digital library and search engine that focuses primarily on the literature in computer and information science. Since its inception, the original CiteSeer grew to index over 750,000 documents and served over 1.5 million requests daily, pushing the limits of the system's capabilities. Access is by subscription.

Google Scholar is a freely accessible web search engine that indexes the full text of scholarly literature and indexes most peer-reviewed online journals of Europe and America. It is similar in function to the subscription-based tools: Elsevier's Scopus and Thomson Reuter's ISI Web of Science, CiteSeerX, and getCITED. Springer-Link covers Springer publications. Another search engine, Microsoft Academic Search (http://academic.research.microsoft.com/) indexes almost ten million publication and six million authors. Journals are also indexed in Academic OneFile, Computer Abstracts International Database, Computer Science Index, Digital Mathematics Registry, Journal Citation Reports/Science Edition, Mathematical Reviews, Science Citation Index Expanded (SciSearch), SCOPUS, VINITI - Russian Academy of Science, Zentralblatt Math.

Guides to computer science materials and literature can be found on Wikipedia. For example, AcademicInfo is an online reference for researchers, and provides an extensive list of computer science digital libraries at (http://www.academicinfo.net/compscilibrary.html) including the Directory of Computing Science Journals. Hypatia (http://www.hypatia-trust.org.uk/library.html) is a directory of researchers in computer science and mathematics, and a library of their papers.

The collection of computer science bibliographies prepared by Alf-Christian Achilles and Paul Ortyl contains more than three million of references (mostly to journal articles, conference papers and technical reports), clustered in about 1,500 bibliographies, and consists of more than 2.3 GB (530 MB gzipped) of BibTeX entries. More than a million references contain URLs to an online version of the paper. (http://liinwww.ira.uka.de/bibliography/index.html).

A unique resource is the Charles Babbage Institute, Founded by Erwin and Adelle Tomash in 1978 and moved to the University of Minnesota in 1979. CBI archivists collect, preserve, and make available for research primary source materials relating to the history of information technology. The archival collection consists of corporate records, manuscript materials, records of professional associations, oral history interviews, trade publications, periodicals, obsolete manuals and product literature, photographs, films, videos, and reference materials. In an Oral History interview, for example, Bruce H. Barnes describes his duties as a program director at the NSF, with examples of NSF's support of research in theoretical computer science, computer architecture, numerical methods, and software engineering, and the development of networking. He describes NSF's support for the development of computing facilities through the Coordinated Experimental Research Program.

Dictionaries, Encyclopedias and Tutorials

Other CS-related online publications include dictionaries, encyclopedias and tutorials, some of which have been mentioned in the section on Education. Among the online publications are numerous free searchable dictionaries for computer and Internet technology definitions, and abbreviations, such as:

- Free On-Line Dictionary of Computing (FOLDOC) jargon, programming languages, and theories related to computing. It contains over 13,000 entries which are cross-referenced to each other and to related resources elsewhere on the web.
- Dictionary of Algorithms and Data Structures covering algorithmic techniques, data structures, archetypical problems, and definitions related to computer science.
- BABEL, a glossary of computer-related acronyms and abbreviations.
- Chip Directory providing numerically and functionally ordered chip lists, chip pinouts, and lists of manufacturers.

Encyclopedias include:

- TechWeb Encyclopedia, a free, online encyclopedia of over 20,000 IT terms.
- Symantec's Virus Encyclopedia, providing synopses of the latest virus-related threats including technical details about how each functions, and instructions for removal. Several universities and other agents produce online tutorials and courses, such as the prestigious MIT and UCLA lectures. Others include:
- W3 Schools Online Web Tutorials introducing web design and development through HTML, XML, browser scripting, server scripting, and multimedia.
- UMBC AgentWeb introducing software agent-related concepts and technologies.

Statistics Part 3: Funding

Funding for computer science research primarily comes from the Federal government through the NSF (see Computer & Information Science and Engineering: CISE at http://www.nsf.gov/dir/index.jsp?org=CISE), the Department of Energy, Department of Defense, or through Pell or other grants that provide student loans. The NSF accounts for approximately 20% of federal support to academic institutions for basic research. Corporate giving and private philanthropy help build and support computer science programs and institutes. UNESCO is also interested in supporting computer science and informatics. Professional organizations such as CRA work to influence government opinion towards increased support of computer science (see Computing Research Policy Blog at http://www.cra.org/govaffairs/blog/).

Federal and State Funding

In November 2010, Director of Government Affairs for the CRA Peter Harsha reported on a draft review of the federal government's 14 agency, $4 billion a year, Networking and Information Technology Research and Development (NITRD) program, which calls for significant new investment in federal IT research support, the establishment of a standing committee of networking and IT specialists to oversee the federal effort, and the establishment of a new, publicly-accessible, detailed database on federal IT research spending. The report calls for new research in high performance computing, privacy and confidentiality, human-computer interactions, large scale data analytics, and cyber physical systems. The review found that NITRD was successful, but also found several issues with the program. For example, while several agencies (such as the Department of Defense) clearly understand the importance of fundamental computing research to their agency missions, many others still don't. Some of this can be seen in the way agencies report their IT research spending levels, mistaking investments in IT infrastructure as investments in IT research. A review by the subcommittee of the funding levels for "IT research" reported by the National Institutes of Health ($1.2 billion in FY 10), for example, showed that true IT research accounted for only 2–11% of the total. And NIH isn't alone. Co-chair of the report committee and chair of CCC Ed Lazowska, says that the NITRD significantly overstates the total federal investment. Also, while NITRD coordinates efforts well, there is little vision and leadership.

The NSF/Division of Science Resources Statistics, Survey of Federal Funds for Research and Development has extensive tables showing annual federal obligations for research in mathematics and computer sciences, by agency and field. Federal obligations for research in computer science and mathematics across all agencies listed for 2007 was almost three billion dollars, with computer science receiving a little over two billion and mathematics a little less than one billion. Several of the agencies had no listing for computer science research (Smithsonian, Social Security, US Census, Housing and Urban Development).

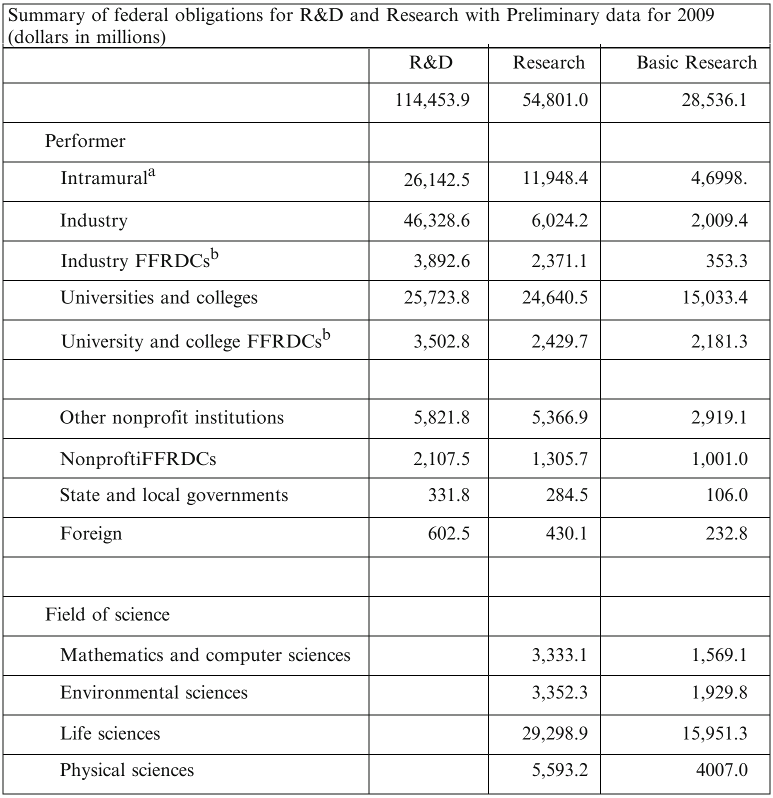
In 2007, the Department of Defense received the largest share of computer science research dollars (about $700 million) with about half going to the Defense Advanced Research Projects Agency, and lesser amounts to the Army, Air Force, and Navy (respectively). The second largest expenditure (about $670 million) went to the Department of Energy, mostly to the National Nuclear Security Administration, and about $53 million of that to the Office of Science. The third largest expenditure went to the NSF (about $600 million). The Department of Commerce received about $63 million for the National Institute of Standards and Technology and $2 million for the National Oceanic and Atmospheric Administration.

Other agencies received lesser amounts (in millions, approximately): NASA (16.5), US Geological Survey (12), Department of Transportation (9), Homeland Security Science and Technology Directorate (8), Environmental Protection (6), and Forest Service (1), and Federal Communications Commission (0.4). Health and Human Services mostly received about 8.5 for the Agency for Healthcare Research and Quality and about 0.5 for Centers for Disease Control and Prevention.

The same NSF survey has tables showing federal obligations for research and development and R&D plants, as well as amounts for research and basic research. A portion of Table 2 showing Federal obligations for R&D and Research Preliminary data for 2009 is presented in Fig. 17.9, with amounts for environmental sciences, life sciences and physical sciences included to show contrast.

The Federal obligations for basic research in computer science, as compiled by the NSF, increased from $438 M in 1999 to $730.5 M in 2003 (Fig. 17.10). There was a decrease in 2005 to $658 M, followed by an increase in 2007 to $708 M. Data for 2009 has not yet been made available.

In 2007–2008, public institutions spent $261 billion ($27,176 per student in 2008–2009 dollars). About 28% of this amount, $7,703 per student, was spent on instruction. About 10% of the remaining funds were used for research (not specifically computer science research), about $2700/FTE student. Private not-for-profit



Summary of federal obligations for R&D and Research with Preliminary data for 2009 (dollars in millions)

	R&D	Research	Basic Research
	114,453.9	54,801.0	28,536.1
Performer			
Intramural[a]	26,142.5	11,948.4	4,6998.
Industry	46,328.6	6,024.2	2,009.4
Industry FFRDCs[b]	3,892.6	2,371.1	353.3
Universities and colleges	25,723.8	24,640.5	15,033.4
University and college FFRDCs[b]	3,502.8	2,429.7	2,181.3
Other nonprofit institutions	5,821.8	5,366.9	2,919.1
NonproftiFFRDCs	2,107.5	1,305.7	1,001.0
State and local governments	331.8	284.5	106.0
Foreign	602.5	430.1	232.8
Field of science			
Mathematics and computer sciences		3,333.1	1,569.1
Environmental sciences		3,352.3	1,929.8
Life sciences		29,298.9	15,951.3
Physical sciences		5,593.2	4007.0

Fig. 17.9 Summary of federal obligations for research with preliminary data for 1999
Source: Table 25: federal obligations for research in mathematics and computer sciences and in social sciences, by agency and detailed field: FY 2007 national science foundation/division of science resources statistics, survey of federal funds for research and development: FY 2007, 2008, and 2009. (http://www.nsf.gov/statistics/nsf10305/content.cfm?pub_id=3966&id=2)

institutions spent $134 billion ($44,592 per student), and 11% of total expenses went towards research ($4835/FTE student).

(Source: Indicator 49, Table A-49-2, Postsecondary Revenues and Expenses, in Contexts of Postsecondary Education Section 5. U.S. Department of Education, National Center for Education Statistics, 2007–2008 Integrated Postsecondary Education Data System (IPEDS), Spring 2009)

The Taulbee Survey also collects information about external funding for CS research. A decrease in external funding from 2003 to 2006 is reported in Fig. 17.11, however, the number of departments responding to the CRA survey also decreased.

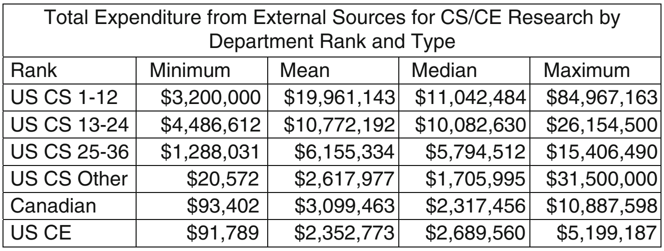
The expenditure of external funding for research in CS varies by rank of department, as shown in Fig. 17.12, part of Table 24.1 created by CRA.

FY 1999–2009 (Dollars in millions, rounded)						
	1999	2001	2003	2005	2007	2009
All fields	17,450	21,958	24,751	27,140	26,866	28,536
Mathematics and computer sciences	735	958	1,120	1,228	1,294	1,569
Computer sciences	438	586	731	658	708	NA
Mathematics	254	288	360	543	561	NA
Mathematics and computer sciences, nec	43	84	30	28	24	NA
Physical sciences	3,090	3,327	3,454	3,739	3,545	4,007
NOTES: Between FY 2006 and FY 2007, National Aeronautics and Space Administration's (NASA) R&D obligations decreased for two reasons: 1) In FY 2007 NASA excluded projects that were operational in nature that were not excluded in FY 2006; and 2) there was an overall decrease in obligations between FY 2006 and FY 2007. In FY 2000, NASA reclassified Space Station as a physical asset, reclassified Space Station Research as equipment, and transferred funding for the program from R&D to R&D plant; and National Institutes of Health reclassified all its development activities as research.						

Fig. 17.10 Federal obligations for research 1999–2009
Source: Table 2: federal obligations for research in mathematics and computer sciences and in social sciences, by agency and detailed field 1999–2009. FY 2007 national science foundation/division of science resources statistics, survey of federal funds for research and development: FY 2007, 2008, and 2009. (http://www.nsf.gov/statistics/nsf10305/content.cfm?pub_id=3966&id=2)

	2003		2006	
	131 departments		126 departments	
	Total	% of funding	Total	% funding
NSF	354,451,309	40.70	255,089,816	43
DARPA	85,401,891	9.80	64,191,150	11
NIH	5,864,767	1.80	24,880,112	4
DOE	20,471,676	2.40	24,391,329	4
State agencies	24,438,483	2.80	16,875,578	2.8
Industrial sources	70,813,388	8.10	50,333,039	8.5
Other defense	177,357,598	20.40	97,512,961	16
Other federal	50,555,980	5.80	32,388,664	5.5
Private foundation	32,977,093	3.80	10,826,656	2
Other	37,995,002	4.40	16,996,108	3
Total	870,327,187		593,485,413	

Fig. 17.11 CRA comparison of CS external funding 2000–2006
Source: Portion of Table 44a. Computing research association 2005–2006 Taulbee survey (www.cra.org)



Total Expenditure from External Sources for CS/CE Research by Department Rank and Type				
Rank	Minimum	Mean	Median	Maximum
US CS 1-12	$3,200,000	$19,961,143	$11,042,484	$84,967,163
US CS 13-24	$4,486,612	$10,772,192	$10,082,630	$26,154,500
US CS 25-36	$1,288,031	$6,155,334	$5,794,512	$15,406,490
US CS Other	$20,572	$2,617,977	$1,705,995	$31,500,000
Canadian	$93,402	$3,099,463	$2,317,456	$10,887,598
US CE	$91,789	$2,352,773	$2,689,560	$5,199,187

Fig. 17.12 CRA comparison of CS research funding by department rank
Source: Table 24.1. Total expenditure from external sources for CS/CE research by department rank and type. CRA 2005–2006 Taulbee survey (www.cra.org). In tables that group departments by rank, the groupings are based on the 1993 National Research Council ranking of research-doctorate programs in the United States, released in 1995 (http://cra.org/statistics/nrcstudy2/)

(USD in millions)	All R&D			Federal R&D		
	2003	2005	2007	2003	2005	2007
All industries	200,724	226,159	269,267	17,798	21,909	26,585
Non-manufacturing industries	79,866	67,969	81,790	4,665	6,274	8,415
Computers and electronic products	39,001	48,296	58,599	6,506	8,522*	8,838
Software	*	16,926	*	*	33	*
Professional/S&T services, incl R&D	27,967	32,021	40,533	4,237	5,839	7,608

*Data have been suppressed by the source to prevent disclosure of confidential information.

Fig. 17.13 Funding of industrial R&D in the USA by major industry, 2003, 2005 and 2007
Source: National science foundation, division of science resources statistics. 2010. *Federal funds for research and development: fiscal years 2007–2009*. Detailed statistical tables NSF 10–305. Portion of Table 125. Arlington, VA. (http://www.nsf.gov/statistics/nsf10305/)

UNESCO Science Report 2010

In 2010, UNESCO published, "The Current Status of Science Around the World". The following international comparisons with U.S. R&D funding, as well as U.S. federal/corporate funding comparisons, and patent information come from that report.

The U.S. consistently invests more money in R&D than the rest of the G8 countries combined. Its share of G7 (excluding Russia, since its budget was not revealed) expenditure on R&D has exceeded 50% since 1997. In 2006, the U.S. share of the G8 total was 53%.

The National Science Foundation has compared (1990–2006) the gross GERD expenditures of U.S. federal and corporate R&D (in constant year 2000 US$M). While federal funding has been flat (from $75B to $82B), total corporate R&D funding has doubled from $102B to $204B. In 2007, R&D federal funding was $93B, and industry funding had grown to $245B. The funding provided by corporations is usually linked to the type of corporate output, and is seldom "pure research". Thus, there is some concern among computer scientists, as well as other scientists and journal publishers about compromises in research integrity due to potential conflicts of interest between scientific researchers and corporate funders.

The prospects for increased R&D investment by business also look bright in many of the emerging scientific nations. Between 2002 and 2007, business investment as a proportion of GDP rose rapidly in China, India, Singapore and South Korea (although India's increase was from a low base); investment has risen rapidly in Japan as well.

The UNESCO Institute for Statistics ranks the annual R&D funding among the top 25 global corporations. In 2003, the ranking included three infotech companies: Microsoft (7th, $6.2B), Intel (13th, $4.4B), and Hewlett-Packard (25th, $3.7B). While U.S.federal funding has gradually declined, corporate R&D funding has increased dramatically: in 2006, Microsoft, for example, invested US$7.8B in R&D, the highest of any multinational corporation. Others headquartered in the US with significant expenditure were IBM ($5.7B), and Intel ($4.8B).

However, the ratio of gross expenditures for research and development (GERD) to gross domestic product (GDP) has never reached 3% in the US. This ratio peaked at 2.9% in 1962. Other countries have exceeded the 3% ratio: Republic of Korea (3.4%) 2008; Japan (3.7%) and Finland (3.5%) in 2007; Israel (4.7%) and Sweden (3.6%) in 2005. Approximately US$ 18.0B (4.9%) of total US GERD was not generated in 2006 by either industry or federal source; this was generated mainly by colleges and universities from their own state funds (US$ 9.9 billion) and other non-profit organizations (US$ 8.1 billion). As the U.S. economy has recently declined, the percentages of funding by states have also suffered declines.

The U.S. still has the largest basic research budget in the world, and an impressive trade surplus in intellectual property, the basis for the innovation needed in business. How do basic research and publications translate into patents? In patent offices, America dominated, with almost 42% of the world's patents in 2006, a share that has fallen only slightly over the previous four years. Japan had about 30%, the EU 26%, South Korea 2% and China 0.5%.

Between 1995 and 2002 in the U.S., the number of university-held patents increased substantially, as did the royalties derived from leasing those patents to industry. Median net royalties grew from $600,000 in 2002 to $900,000 in 2005, although annual patent numbers peaked at 3,300 in 2004, showing that licensing is an important factor in the continued growth of industry-university partnerships begun in the 1970s. As the strongest trend in industry in 2010 is the outsourcing and off-shoring of "open innovation" R&D, the improving universities in China and the former Soviet Union will pull industry investment from the U.S. and the EU, but U.S. intellectual properties are expected to keep their value to industry.

Private Funding for Computer Science

Many universities have institutes for various aspects of computing, but there is no separate entity as in other fields. In 2010 the Simons Foundation announced that it will provide up to $6 M/year funding for a new Institute for the Theory of Computing. The funding recognizes the deep importance of the study of computation to society, and the need for a critical mass of researchers from around the world to accelerate fundamental research on computation and to further develop its interactions with other areas of science ranging from mathematics and statistics to biology, physics and engineering.

The Bill & Melinda Gates Foundation is one of the world's largest philanthropic organizations, and has donated enormous sums to promote literacy in every area (in addition to health and overcoming poverty). For example; the University Scholars Program established at Duke University in 1998 provides scholarships to students in the Graduate School pursuing doctoral degrees in any discipline. The Cornell University Faculty of Computing and Information Science received $25 million for a new Information Science building, expected to be finished by 2014. The Carnegie Mellon School of Computer Science received $20 million for a new Computer Science building, which opened 2009.

The U.S. has a tradition of philanthropic funding excelled nowhere else in the world. Many corporations and foundations give funding to universities and institutions, while others directly support individuals. In addition to many companies and organizations small and large, some of the generous givers include the Alfred P. Sloan Foundation, the Ford Foundation, The John D. and Catherine T. MacArthur Foundation, AT&T, 3 M, Exxon, the Carnegie Foundation, NEC Foundation of America, Sigma Xi, The Scientific Research Society, the Alexander von Humboldt Foundation, the Google Anita Borg Memorial Scholarship, and the L'Oreal USA Fellowships for Women In Science (www.lorealusa.com).

The U.S. has been and continues to be the world leader in education, publishing and patents, employment and funding of computer science. Computer science theory, software and hardware are used in almost every field. The professional agencies, private organizations and government recognize the essential relevance of the study of computation to society, and are working together to ensure that our leadership in innovation and computer science production remain stellar.

Statistics Part 4: Employment

The U.S. Bureau of Labor Statistics (BLS) started developing long-term employment projections nearly 60 years ago, soon after World War II ended. The 2008–2018 projection was released in November 2009 (see Fig. 17.14). The projections are based on a macroeconomic model of the US economy that solves a system of 543 equations. The data baseline was in 2008, which had 7.2% unemployment (and deeper unemployment occurred in 2009 and 2010). The model makes projections under the assumption that the economy will return to full employment and a long-run growth path with yearly average 2.4% GDP by 2018 (and no further interruptions to the economy). Because of the baseline and assumptions, some of the projections may reflect stronger growth than one might expect. The projected job openings are a

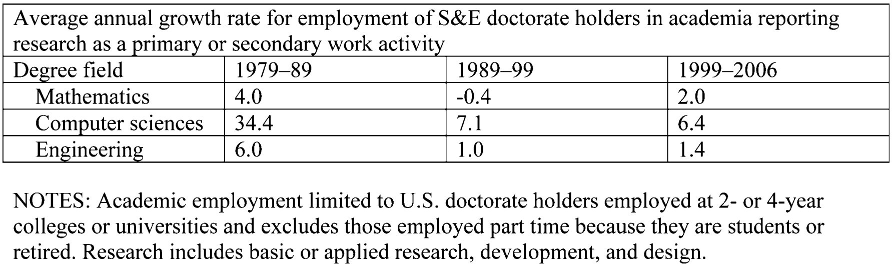
	Number (thousands)			Change 2008-18	Job openings due to growth and replacement needs
National Employment Matrix Title	2008	2018	2018 - 2008		
Computer and mathematical science occupations	3,540.4	4,326.1	785.7	22.2	1,440.5
Computer specialists	3,424.3	4,187.0	762.7	22.3	1,383.6
Computer and information scientists, research	28.9	35.9	7.0	24.2	13.2
Computer Programmers	426.7	414.4	(12.3)	–2.9	80.3
Computer software engineers	909.6	1,204.8	295.2	32.5	371.7
Computer software engineers, applications	514.8	689.9	175.1	34.0	218.4
Computer software engineers, systems software	394.8	515.0	120.2	30.4	153.4
Computer support specialists	565.7	643.7	76.0	13.8	234.6
Computer systems analysts	532.2	640.3	108.1	20.3	222.8
Database administrators	120.4	144.7	24.3	20.3	44.4
Network and computer systems administrators	339.5	418.4	78.9	23.2	135.5
Network systems and data communications analysts	292.0	447.8	155.8	53.4	208.3
All other computer specialists	209.3	236.8	27.5	13.1	72.6
Operations research analysts	63.0	76.9	13.9	22.0	32.2

Note: Total job openings represents the sum of employment increases and replacement needs. If employment change is negative, job openings due to growth are zero and total job openings equals replacement needs.

Fig. 17.14 Current and projected employment in computer science occupations
Source: *Occupational employment projections to 2018* by T. Alan Lacey and Benjamin Wright, Monthly Labor Review, November 2009

measure of the total number of workers who will be needed to meet demand, and include new jobs created from economic growth, plus jobs created by retirement or other replacement. The expectation is that replacement needs will account for more than twice as many 2018 job openings as economic growth.

During the coming 8 years, the 55-and-older group will be a larger share of the U.S. population. As a result, computing occupations related to healthcare services are expected to increase rapidly. As a group, computer and mathematical occupations are projected to grow more than twice as fast as the average for all occupations in the economy and are expected to add a total of 1,440,500 jobs – including 785,700 new jobs – from 2008 to 2018. Computer specialists will account for the vast majority of this job growth, increasing by 762,700 new jobs for a total of 1,383,600 new and replacement CS job openings. Computer software applications engineers will increase by roughly 175,000 new jobs – more than the projected increase for any other type of computer specialists. Network systems and data communications analysts are projected to see an increase of 155,800 new jobs, while other computer systems analysts will be needed in almost 110,000 new jobs, as well as about 167,000 replacement analyst jobs. New computer specialist jobs will arise in almost every



Average annual growth rate for employment of S&E doctorate holders in academia reporting research as a primary or secondary work activity			
Degree field	1979–89	1989–99	1999–2006
Mathematics	4.0	-0.4	2.0
Computer sciences	34.4	7.1	6.4
Engineering	6.0	1.0	1.4

NOTES: Academic employment limited to U.S. doctorate holders employed at 2- or 4-year colleges or universities and excludes those employed part time because they are students or retired. Research includes basic or applied research, development, and design.

Fig. 17.15 Average annual growth rate for employment of S&E doctorate holders in academia reporting research as a primary or secondary work activity

industry, but roughly half will be located in the computer systems design industry, which is expected to employ more than one in four computer specialists in 2018.

According to the U.S. Department of Labor (DoL), computer and information researchers held about 28,900 jobs in 2008. About 23% of these were in computer systems design and related services. They were also employed by software publishing firms, scientific research and development organizations and in education. Researchers categorized by the DoL Standard Occupational Code (SOC) 15–1011 are generally expected to hold the Ph.D. The Occupational Information Network (O*NET) provides information on occupational characteristics. See http://www.bls.gov/ooh/ocos304.htm

Using the Department of Education IPEDS data mentioned earlier, the total number of computer science doctorates produced in the past almost 30 years (1980–2009) is about 22,000. Thirty years is approximately the length of a normal working career, so we can assume that 22,000 is an upper bound on the numbers of Ph.D. computer scientists today. Not every doctorate from the past 30 years will be working in research today, so the gap of almost 10,000 jobs notable when comparing the IPEDS data with the 28,900 researchers reported by the DoL for 2008, suggests that many who hold Master's or Bachelor's degrees hold an occupation title of "researcher".

Employment by Specialty

Employment of new Ph.D.s by specialty in the U.S. and Canada is tracked by the CRA Taulbee Surveys, and this information is summarized the tables in Fig. 17.17. In 2007, AI/robotics took over from OS/networks as the area with the largest number of graduates. The numbers vary only slightly, and so it is difficult to notice trends. In 2007, the choice of areas that the departments could use to classify Ph.D. recipients was refined, and additional categories of interest were added.

The Fig. 17.16 shows the employment of new CS/CE Ph.D.s in industry, government or self-employed versus academia and also those who have gone outside North America. There have been dramatic reversals as in 2001–2003

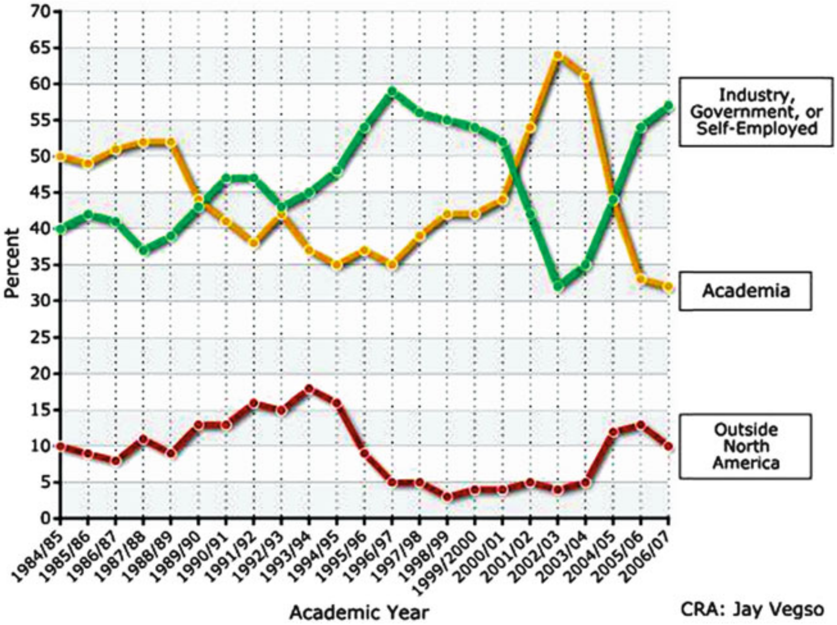


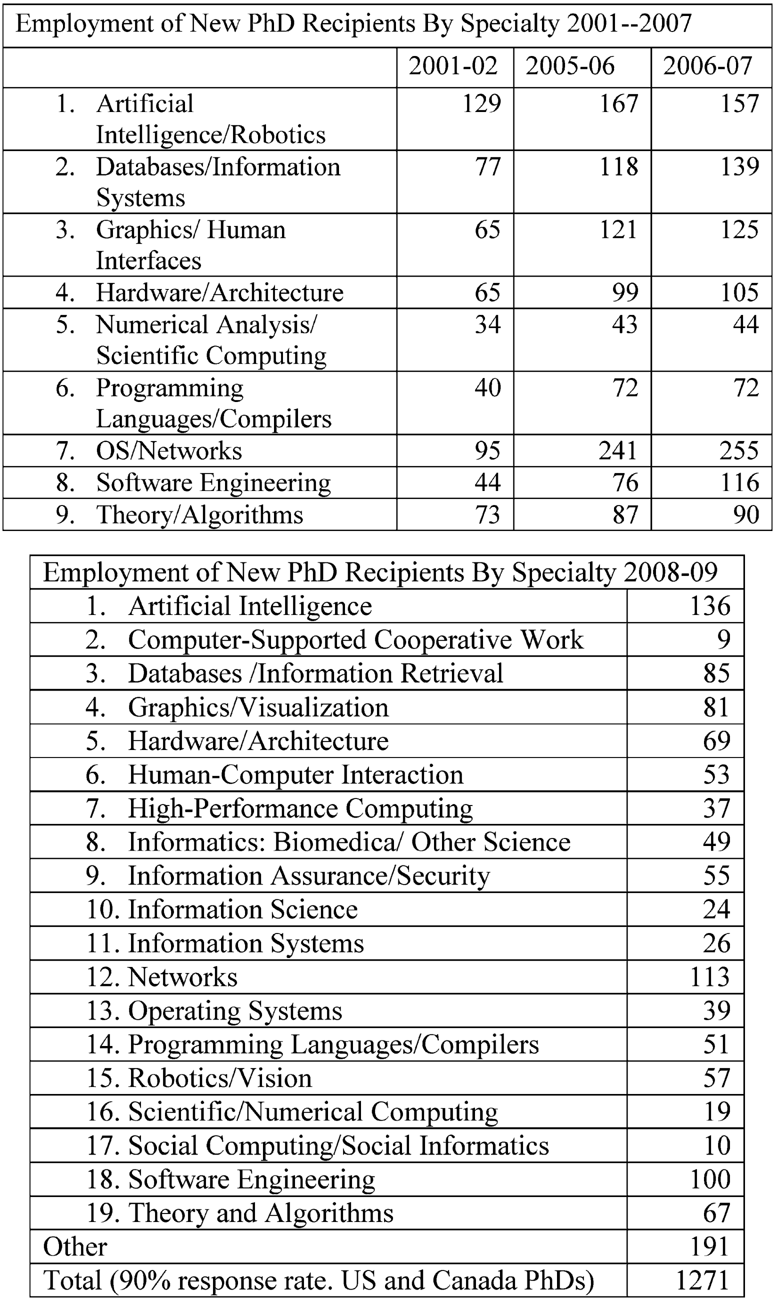
Fig. 17.16 Employment of new PhDs in industry, government, academia, offshore
Source: 2008–2009 CRA Taulbee survey

when doctorates going into academia jumped to 64% from 44%, with those in industry dropping to 32%. By 2006, there was another reverse with industry claiming almost 60% of doctorates. Another reverse may be happening. Only about 47% of 2008–2009 graduates joined industry.

This may be due to the economy. According to outplacement company Challenger, Gray and Christmas, degrees in engineering and computer science, which used to be considered surefire paths to employment, each received just 10% of recruiter votes for offering the best chance of job-search success. The technology sector announced almost 175,000 planned job cuts in 2009, which is 12% higher than the cuts of 2008. The list of those announcing cuts include Google, Microsoft, IBM, Adobe, Yahoo, AOL, AT&T, Sprint, Cisco Systems, Nokia, Seagate and Sun. However, these cuts include sales, marketing and recruiting forces, and not necessarily research. According to CRA, the unemployment rate for new PhDs is less than 1%.

(Source: Joseph Tucci of EMC-Technology Section-Business Management US. www.busmanagement.com GDS Publishing Ltd. 2010)

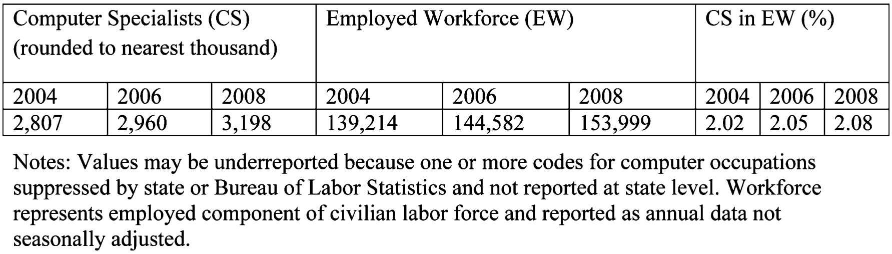
A similar number of graduates took tenure-track jobs in 2008–2009 as did in 2007–2008. However, more graduates went into academic positions as researchers and post-docs in 2008–2009. The new NSF Computing Innovation Fellows



Employment of New PhD Recipients By Specialty 2001--2007			
	2001-02	2005-06	2006-07
1. Artificial Intelligence/Robotics	129	167	157
2. Databases/Information Systems	77	118	139
3. Graphics/ Human Interfaces	65	121	125
4. Hardware/Architecture	65	99	105
5. Numerical Analysis/ Scientific Computing	34	43	44
6. Programming Languages/Compilers	40	72	72
7. OS/Networks	95	241	255
8. Software Engineering	44	76	116
9. Theory/Algorithms	73	87	90

Employment of New PhD Recipients By Specialty 2008-09	
1. Artificial Intelligence	136
2. Computer-Supported Cooperative Work	9
3. Databases /Information Retrieval	85
4. Graphics/Visualization	81
5. Hardware/Architecture	69
6. Human-Computer Interaction	53
7. High-Performance Computing	37
8. Informatics: Biomedica/ Other Science	49
9. Information Assurance/Security	55
10. Information Science	24
11. Information Systems	26
12. Networks	113
13. Operating Systems	39
14. Programming Languages/Compilers	51
15. Robotics/Vision	57
16. Scientific/Numerical Computing	19
17. Social Computing/Social Informatics	10
18. Software Engineering	100
19. Theory and Algorithms	67
Other	191
Total (90% response rate. US and Canada PhDs)	1271

Fig. 17.17 Employment of new Ph.D. Recipients by specialty
Source: Table 4 from Taulbee surveys. CRA archives



Computer Specialists (CS) (rounded to nearest thousand)			Employed Workforce (EW)			CS in EW (%)		
2004	2006	2008	2004	2006	2008	2004	2006	2008
2,807	2,960	3,198	139,214	144,582	153,999	2.02	2.05	2.08

Notes: Values may be underreported because one or more codes for computer occupations suppressed by state or Bureau of Labor Statistics and not reported at state level. Workforce represents employed component of civilian labor force and reported as annual data not seasonally adjusted.

Fig. 17.18 Computer specialists as share of workforce
Source: Table 8.32. Computer specialists as share of workforce, by state: 2004, 2006, and 2008. Bureau of Labor Statistics, Occupational employment and wage estimates; and local area unemployment statistics. *Science and engineering indicators 2010*

program had a lot to do with supporting this shift. (See http://www.cra.org/uploads/documents/resources/taulbee/0809.pdf). The CRA Taulbee Survey annually reports on numbers of academic open positions and hires, by ranked departments.

Foreign-Born S&E Academics

Foreign-born S&E doctorate holders with U.S. doctorates are more heavily concentrated in computer sciences, mathematics, and engineering than in other fields. These foreign-born doctorate holders account for more than half of all academic researchers and of full-time faculty researchers in computer sciences and for 39–48% of all academic researchers and full-time faculty researchers in mathematics and engineering. In contrast, they represent 27% or less of all academic researchers and 21% or less of full-time faculty researchers in the life sciences, the physical sciences, psychology, and the social sciences.

(Source: Chap. 5. Academic Research and Development. Doctoral Scientists and Engineers in Academia, *NSF Science and* Engineering *Indicators 2010*. (http://www.nsf.gov/statistics/seind10/c5/c5s3.htm)).

Computing in Industry

The U.S. Bureau of Labor Statistics, Occupational Employment and Wage Estimates, shows that the occupation "Computer Specialist" represents about 2% of the total US workforce (see Fig. 17.18). The data are reported state-by-state based on DoL statistics and state unemployment data.

Employment in high-technology establishments as share of total employment has remained about 11.5% during 2003, 2004, and 2006.

(Source: Table 8.48: Employment in high-technology establishments as share of total employment, by state: 2003, 2004, and 2006. Census Bureau, 1989–2006 Business Information Tracking Series, special tabulations. Science and Engineering Indicators 2010)

About 80% of workers with S&E doctorates in 2006 work in jobs that are closely or somewhat closely related to their degrees, as compared with about 75% of those holding Master's, or 60% of those with Bachelor's degrees.

(Source: Table 3.4. Individuals with highest degree in S&E employed in S&E-related and non-S&E occupations, by highest degree and relationship of highest degree to job: 2006. NSF, Division of Science Resources Statistics, Scientists and Engineers Statistical Data System (SESTAT)(2006), http://sestat.nsf.gov. Science and Engineering Indicators 2010.)

Basic research in computer science algorithms and theory is part of the research and development of companies such as Google, Yahoo, Bell Labs, FedEx, IBM, Intel, Microsoft, AT&T, and other corporations. Their private research labs support student interns, post-docs, sponsor academic conferences and workshops, and they help advocate for government policy in support of computer science. The amount of basic research is difficult to separate out from development, since activity is motivated by potential products as well as by the academic research community.

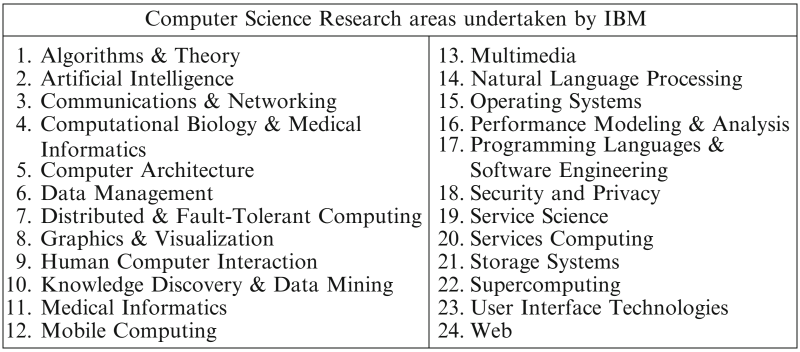
For example, Bell Labs, the research arm of Alcatel-Lucent, includes the Computing and Software Principles Research Department, part of the Enabling Computing Technologies domain. The department performs fundamental research in both systems and theoretical areas, driven by real-world problems needing answers found in algorithms, database systems, formal methods, and telecom and web services (Source: Bell Labs website).

The Computer Science website at IBM Research - Almaden lists 24 areas of CS research, and proposes to lead the next generation of research in database management, intelligent information systems, user productivity, healthcare IT and the theoretical foundations of computer science.

Microsoft Research has almost a thousand researchers, including computer scientists, sociologists, psychologists, mathematicians, physicists, and engineers, working across more than 60 disciplines. In addition to the areas above, Microsoft adds Gaming, Information Retrieval, Machine Learning, and Social Science and Computation as CS research focal areas.

Google and Yahoo have a similar list of research areas, and also add cryptography, hypertext and the Web, economics, video processing and virtual reality.

FedEx has seven IT centers across the U.S., including FedEx Labs and the FedEx Institute of Technology at the University of Memphis. FedEx has over 7,000 IT professionals with more than 275,000 employees worldwide. FedEx research has allowed it to track over ten million packages on a single day, while posting to databases approximately 3,000 transactions per second on shipment movement,



Computer Science Research areas undertaken by IBM

1. Algorithms & Theory
2. Artificial Intelligence
3. Communications & Networking
4. Computational Biology & Medical Informatics
5. Computer Architecture
6. Data Management
7. Distributed & Fault-Tolerant Computing
8. Graphics & Visualization
9. Human Computer Interaction
10. Knowledge Discovery & Data Mining
11. Medical Informatics
12. Mobile Computing
13. Multimedia
14. Natural Language Processing
15. Operating Systems
16. Performance Modeling & Analysis
17. Programming Languages & Software Engineering
18. Security and Privacy
19. Service Science
20. Services Computing
21. Storage Systems
22. Supercomputing
23. User Interface Technologies
24. Web

Fig. 17.19 Computer Science Research undertaken by IBM
Source: IBM website at: //researcher.ibm.com/researcher/ (Sighted 07/10/11)

while simultaneously responding to a thousand inbound inquiries on package status information.

(Source: *FedEx think-tanks-Perfect Package*, by Leslie Knudson, Deputy Editor Issue 9 Business Management e-magazine June 2007. http://www.busmanagement.com/article/Perfect-package/ viewed October 2010.)

The estimated share of computer-related services in company-funded R&D and domestic net sales of R&D-performing companies: 1987–2007 has climbed from about 3.8% in 1987 to about 14% in 2007.

(Source: Table 4.6 of NSF, Division of Science Resources Statistics, Survey of Industrial Research and Development (annual series), http://www.nsf.gov/statistics/srvyindustry, Viewed 6 May 2009. *Science and Engineering Indicators 2010*)

Prestige of the Occupation of "Scientist"

The Harris Poll has rated the prestige of various occupations. The occupation of "scientist" ranked 66 (high prestige) in 1977, decreased to 51 in 2002, and increased to 56 in 2008. These rankings are higher than most any other occupation (about the same as Firefighter and Doctor).

NOTES: Responses to the interviewer saying, "I am going to read off a number of different occupations. For each, would you tell me if you feel it is an occupation of very great prestige, considerable prestige, some prestige, or hardly any prestige at all?"

(Source: Prestige Paradox: High Pay Doesn't Necessarily Equal High Prestige: Teachers' Prestige Increases the Most Over 30 Years, Harris Poll, Harris Interactive (5 August 2008), http://www.harrisinteractive.com/harris_poll/index.asp?PID=939, accessed 22 September 2009. Science and Engineering Indicators 2010.)

The outlook for having enough CS scientists in the U.S. is mixed. There are warnings that the U.S. is facing a severe shortage of skilled IT workers. There are several arguments for this view. There is publicity in the popular press on offshoring of computer work. There is some belief that IT firms mainly hire workers (from India) on H-1B visas because they can be paid less. Popular opinion is that U. S. students, despite or perhaps because of their constant use of electronics, are not interested in computer science, or mathematics studies, and certainly not interested in competing for low wages.

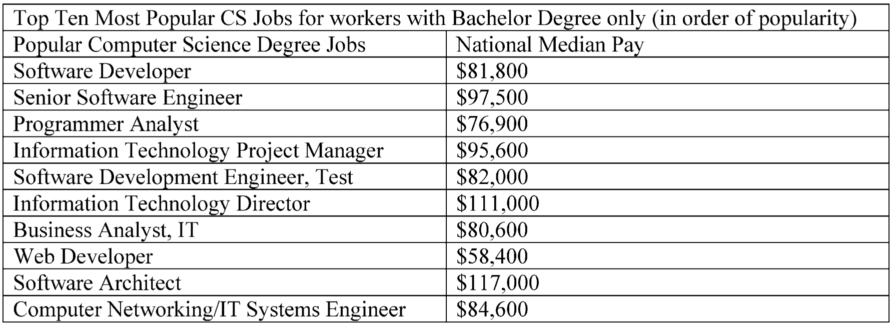
Computer Science Job Prospects in 2010

Engineering, computer science and accounting may no longer be the fastest path to employment, but they are among the most lucrative. A recent survey by the National Association of Colleges and Employers found that eight of the top ten best-paid majors are in engineering, with the highest-paid petroleum engineering graduates starting at $86,220. Computer science ranked fourth in the NACE survey, with graduates earning average starting salaries of $61,205. A high starting salary is no guarantee of job-search success, however. NACE found that only 42% of engineering graduates found jobs in 2009, compared to 70% in 2007.

(Source: Matt Krumrie, Report: 2010 College Graduate Jobs Outlook May 11th, 2010 5:46 pm CT. Minneapolis Workplace Examiner. http://www.examiner.com/workplace-in-minneapolis/report-2010-college-graduate-jobs-outlook)

The figure (Fig. 17.20) shows results from a 2010 survey conducted by PayScale. com, an employment placement agency. All data is limited to those with a Bachelor's degree and no higher degrees who work full-time in the United States. Jobs are listed in order of relative popularity amongst graduates with a Bachelor's degree in the given major from any college. Median salary for each job title is for individuals with any major who have a typical amount of experience at that job. Salary is the sum of compensation from base salary, bonuses, profit sharing, commissions, and overtime, if applicable. Salary does not include equity (stock) compensation.

The sudden economic near-depression in the U.S. has created an unusual depression among 2009 U.S. college graduates: 80% moved back home with their parents after graduation, according to a report by CollegeGrad.com, up from 67% in 2006. A study by Challenger found that many newly-minted graduates were accepting lower-paying service sector positions or forsaking income entirely by volunteering or accepting unpaid internships. Others may abandon the job search, opting to further their education or travel. As IT-focused companies grow in emerging markets around the world, more CS graduates opt to find work in other countries.



Top Ten Most Popular CS Jobs for workers with Bachelor Degree only (in order of popularity)	
Popular Computer Science Degree Jobs	National Median Pay
Software Developer	$81,800
Senior Software Engineer	$97,500
Programmer Analyst	$76,900
Information Technology Project Manager	$95,600
Software Development Engineer, Test	$82,000
Information Technology Director	$111,000
Business Analyst, IT	$80,600
Web Developer	$58,400
Software Architect	$117,000
Computer Networking/IT Systems Engineer	$84,600

Fig. 17.20 Most popular CS jobs for workers with bachelor degree only
Source: http://www.payscale.com/best-colleges/computer-science-degree.asp. Viewed 5 October 2010)

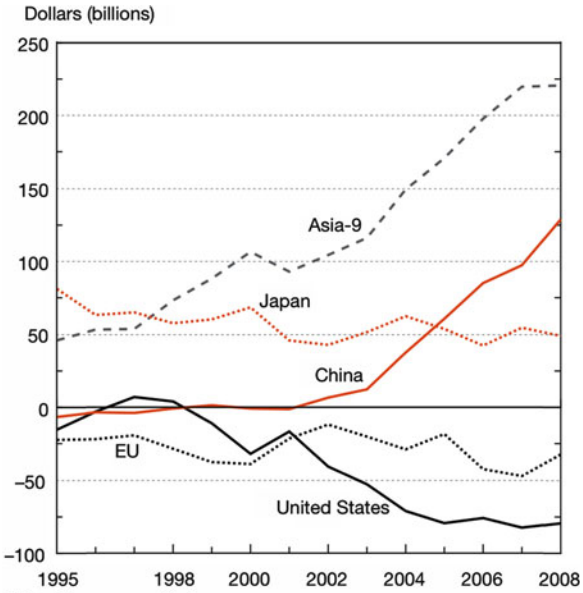
(Source: *College Grads Enter Tough Job Market*. Press Release from Challenger, Gray and Christmas, Inc., Chicago, April 14, 2010. James K. Pedderson, Director of Public Relations.)

Off-Shoring

From the Technology Sector of the online NY Times (September 2010), job growth in fields like computer systems design and Internet publishing has been slow in the last year. Employment in areas like data processing and software publishing has actually fallen. Computer scientists, systems analysts and computer programmers all had unemployment rates of around 6% in the second quarter of this year.

"There's been this assumption that there's a global hierarchy of work, that all the high-end service work, knowledge work, R.&D. work would stay in U.S., and that all the lower-end work would be transferred to emerging markets," said Hal Salzman, a public policy professor at Rutgers and a senior faculty fellow at Heldrich Center for Workforce Development. "That hierarchy has been upset, to say the least," he said. "More and more of the innovation is coming out of the emerging markets, as part of this bottom-up push." This change is indicated in Fig. 17.21, which reflects the change in trade balance in high-technology goods.

In a study involving over 3,000 human resources managers and 6,000 US workers (How Offshoring Affects IT Workers, Communications of the ACM, October 2010) Prasanna Tambe and Lorin Hitt report that IT workers experience offshoring-related displacement at a rate of 8%, more than double that of workers in other occupations. Computer programmers, software engineers, systems analysts and customer service jobs are listed as commonly offshored type of work, primarily for cost reasons.



EU = European Union

Fig. 17.21 Trade balance in high-technology goods for selected regions/countries: 1995–2008. *EU* European union.

Notes: Asia-9 includes India, Indonesia, Malaysia, Philippines, South Korea, Singapore, Taiwan, Thailand, and Vietnam. China includes Hong Kong. EU excludes Cyprus, Estonia, Latvia, Lithuania, Luxembourg, Malta, and Slovenia

Source: Catherine Ratherine. Once a dynamo, the tech sector is slow to hire. September 6, 2010, (Online at http://www.nytimes.com/2010/09/07/business/economy/07jobs.html?_r=1)

The U.S. National Laboratories as Employers of CS Researchers

The United States Department of Energy National Laboratories and Technology Centers are a system of facilities and laboratories overseen by the United States Department of Energy (DOE) for the purpose of advancing science and helping promote the economic and defensive national interests of the United States of America. The national laboratory system, administered first by the Atomic Energy Commission, then the Energy Research and Development Administration, and currently the Department of Energy, is one of the largest (if not the largest) scientific research systems in the world. The DOE provides more than 40% of the total national funding for physics, chemistry, materials science, and other areas of the physical sciences. Many are locally managed by private companies, while other are managed by academic universities, and as a system they form one of the overarching and far-reaching components in what is known as the “iron triangle” of military, academia, and industry.

The United States Department of Energy operates 16 national laboratories:

- Lawrence Berkeley National Laboratory, Berkeley, California (1931)
- Los Alamos National Laboratory, Los Alamos, New Mexico (1943)
- Oak Ridge National Laboratory, Oak Ridge, Tennessee (1943)
- Argonne National Laboratory, DuPage County, Illinois (1946)
- Ames Laboratory, Ames, Iowa (1947)
- Brookhaven National Laboratory, Upton, New York (1947)
- Sandia National Laboratories, Albuquerque, New Mexico and Livermore, California (1948)
- Idaho National Laboratory, between Arco and Idaho Falls, Idaho (1949)
- Princeton Plasma Physics Laboratory, Princeton, New Jersey (1951)
- Lawrence Livermore National Laboratory, Livermore, California (1952)
- Savannah River National Laboratory, Aiken, South Carolina (1952)
- National Renewable Energy Laboratory, Golden, Colorado (1956)
- SLAC National Accelerator Laboratory, Menlo Park, California (1962)
- Pacific Northwest National Laboratory, Richland, Washington (1965)
- Fermi National Accelerator Laboratory, Batavia, Illinois (1967)
- Thomas Jefferson National Accelerator Facility, Newport News, Virginia (1984)

The U.S. Office of Naval Research (ONR) within the United States Department of the Navy coordinates, executes, and promotes the science and technology programs of the U.S. Navy and Marine Corps through schools, universities, government laboratories, and nonprofit and for-profit organizations. Authorized by an Act of Congress, and approved by President Truman in 1946, ONR executes its mission by funding (through grants and contracts) world-class scientists who perform basic research, technology development, and advanced technology demonstrations. More than 50 researchers have won a Nobel Prize for their ONR-funded work. ONR's S&T Portfolio is balanced to meet the broad spectrum of warfare requirements with 40% allocated to Discovery & Invention (Basic and Applied Science).

Naval Research Laboratory (NRL) is the corporate research laboratory for the Navy and Marine Corps and conducts a broad program of scientific research, technology and advanced development. NRL was founded in 1923, and employs roughly 1,500 scientists and engineers.

In June 2006, Los Alamos National Laboratory (approximately $2 billion in annual R&D expenditures in recent years) became industry administered; previously, UC administered. This shift is one reason for change in trends apparent in R&D expenditure figures between 2006 and 2007.

The U. S. Department of Defense, through many agencies and labs in the Army, Navy, Air Force, Marines, Coast Guard, National Guard, Customs and Homeland Security, and other organizations, supports research in computer science. Advanced computing is the backbone of the Department of Defense and of critical strategic importance to our nation's defense. All DoD sensors, platforms and missions depend heavily on computer systems. For example, the Maui High Performance

Computing Center (MHPCC) is an Air Force Research Laboratory Center managed by the University of Hawai'i.

The Defense Advanced Research Projects Agency (DARPA) is the research and development office for the U. S. Department of Defense (DOD). Started in 1958 as a response to the Soviet Sputnik, DARPA funds unique and innovative research through the private sector, academic and other non-profit organizations as well as government labs in order to "prevent and create strategic surprise". DARPA has seven technology offices, including the Transformational Convergence Technology Office (TCTO) which seeks to advance innovation in "new crosscutting capabilities derived from a broad range of emerging technological and social trends, particularly in areas related to computing and computing-reliant subareas of the life sciences, social sciences, manufacturing, and commerce." DARPA initiated the Ubiquitous High Performance Computing (UHPC) program to create a new generation of computing systems, and is developing the ExtremeScale Supercomputer System. Prototype UHPC systems are expected to be complete by 2018, developed by Intel, NVIDIA, MIT CS and AI Laboratory, and Boston and Sandia National Laboratory. Georgia Institute of Technology, Atlanta, was selected to lead a team for evaluating the UHPC systems under development. Core computing research areas of interest for DARPA are:

- Computer systems and architectures
- Networked systems science
- Cybersecurity
- Software systems (systems and languages)
- AI (including robotics and vision).

The National Aeronautics and Space Administration (NASA) was established by President Eisenhower in 1958 with the mission of pioneering non-military space research and exploration. In addition to six Test facilities, six Construction and Launch facilities, and four Deep Space Network facilities, NASA runs six research centers: Ames Research Center at Moffett Federal Airfield in Mountain View, California, the Jet Propulsion Laboratory at California Institute of Technology in Pasadena, California, the Goddard Institute for Space Studies in New York City, the Goddard Space Flight Center in Maryland, the John H. Glenn Research Center at Lewis Field, Ohio, and Langley Research Center in Virginia. In addition to providing employment to research scientists, NASA supports education from K-university. For example, Motivating Undergraduates in Science and Technology program, or MUST, provides summer JPL internships.

National Science Foundation (NSF) founded by congressional act in 1950 provides grants to researchers and research facilities to support all non-medical fields of basic research, and also science, engineering and mathematics education from pre-K through graduate school. One of the NSF seven directorates is Computer and Information Science and Engineering (fundamental computer science, computer and networking systems, and artificial intelligence). The NSF has numerous programs in STEM education at all levels: summer programs for undergraduates (REU), Integrative Graduate Education Research Traineeships (IGERT) for graduate students. Alliance for Graduate Education and the Professoriate (AGEP) programs, Graduate Research Fellowships, and an early career-development program (CAREER). The hope is that these programs and many others will provide the necessary computer scientists to maintain the U.S. position as world technology leader.

Statistics Part 5: Professional Associations

The professional associations have profoundly influenced the promotion and growth of computer science research in many ways. Each association, the CRA especially, seeks to inform and influence government policies and funding. As mentioned in the section on publishing, the associations support workshops, conferences, special interest groups and other means for scientific collaboration and the dissemination of research findings, well as being publishers and archivists of proceedings, journals and bibliographies. The professional associations support students and early professionals with mentoring, scholarships, and employment fairs. Further, the associations provide prizes, awards and other recognition for scholarly accomplishment and service. Below are listed a few of the key professional associations supporting research in computer science.

The Association for Computing Machinery (ACM) was founded in 1947 by leaders in electronic and digital computing machinery. As of 2006, there are 62,000 professional members and 20,000 student members. ACM sets curriculum guidelines and standards. It hosts a Digital Library with leading-edge publications, online books and courses, conferences, student chapters and career resources. One of the most prestigious technical computer science awards is the ACM Turing Award, accompanied by a prize of $100,000. The ACM-W provides the Athena Lecturer Award and works to encourage women in computing.

The ACM has 33 special interest groups (SIGs), each holding specialized conferences, usually with proceedings.

SIGACCESS – Accessible Computing
SIGAda – Ada Programming Language
SIGAPP – Applied Computing
SIGARCH – Computer Architecture
SIGART – Artificial Intelligence
SIGBED – Embedded Systems
SIGCAS – Computers and Society
SIGCHI – Computer-Human Interaction
SIGCOMM – Data Communication
SIGCSE – Computer Science Education

SIGDA – Design Automation
SIGDOC – Design of Communication
SIGecom – Electronic Commerce
EVO – Genetic and Evolutionary Computation
SIGGRAPH – Computer Graphics and Interactive Techniques
SIGIR – Information Retrieval
SIGITE – Information Technology Education
SIGKDD – Knowledge Discovery in Data
SIGMETRICS – Measurement and Evaluation
SIGMICRO – Microarchitecture
SIGMIS – Management Information Systems
SIGMM – Multimedia
SIGMOBILE – Mobility of Systems, Users, Data and Computing
SIGMOD – Management of Data
SIGOPS – Operating Systems
SIGPLAN – Programming Languages
SIGSAC – Security, Audit and Control
SIGSAM – Symbolic and Algebraic Manipulation
SIGSIM – Simulation and Modeling
SIGSOFT – Software Engineering
SIGSPATIAL – SIGSPATIAL
SIGUCCS – University and College Computing Services
SIGWEB – Hypertext, Hypermedia and Web

The Association for the Advancement of Artificial Intelligence (AAAI) was founded in 1979, and now has a digital library of more than 10,000 AI technical papers.

The Computing Research Association (CRA) is an association of more than 200 North American academic departments of computer science, computer engineering, and related fields; laboratories and centers in industry, government, and academia engaging in basic computing research; and affiliated professional societies. A large part of CRA's mission is to influence government policy. CRA hosts an annual Computing Leadership Summit with senior leaders of its six affiliate societies: AAAI, ACM, CACS/AIC, IEEE-CS, SIAM and USENIX and the NRC's Computer Science and Telecommunications Board. CRA gathers extensive data about the field and publishes it through the Taulbee Report. CRA maintains information on the Forsythe List, a list of colleges and universities in US and Canada that offer degrees in computer science and computer engineering. The CRA Career Mentoring Workshop aids graduate students and junior faculty as they choose or begin careers. CRA-W supports women in computing research with mentoring, Anita Borg awards, and the Grace Hopper Celebration of Women in Computing conference. The Anita Borg Institute for Women and Technology promotes CS with scholarships for women, the Grace Hopper Celebration, and Systers – an email mentoring program.

The Computing Community Consortium (CCC) is a consortium of computing experts who act to provide scientific leadership and vision on issues related to computing research and future large-scale computing research projects. The National Science Foundation announced in 2006 an agreement with CRA to establish CCC to help identify major research opportunities and establish "grand challenges" for the field.

The Institute of Electrical and Electronics Engineers–Computer Society (IEEE-CS) was founded in 1946 and with nearly 85,000 members and many student chapters is the largest of the 38 societies of IEEE. The Computer Society offers technical journals, magazines, conferences, books, conference publications, and online courses. The IEEE-CS offers curriculum standards, a Certified Software Development Professional (CSDP) program for mid-career professionals and Certified Software Development Associate (CSDA) credential for recent college graduates, and the CS Digital Library (CSDL) containing more than 250,000 articles from 1,600 conference proceedings and 26 CS periodicals going back to 1988.

The National Academies are comprised of four organizations: the National Academy of Sciences (NAS), the National Academy of Engineering (NAE), The Institute of Medicine (IOM) and the National Research Council (NRC), and the Transportation Research Board is a major unit. The NAS was created in 1863 by a congressional charter approved by President Abraham Lincoln. Under this charter, the NRC was established in 1916, the NAE in 1964, and the IOM in 1970. These private, nonprofit organizations share in the responsibility for advising the federal government, upon request and without fee, on questions of science, technology, and health policy. The NAS, NAE, and IOM are honorific organizations; new members are elected annually, and membership is considered a high honor. The National Academy of Sciences publishes a scholarly journal: *Proceedings of the National Academy of Sciences*.

While neither the National Sciences Board (NSB) nor the National Science Foundation is a professional association, they both are significant to research. The NSB was created by congressional act in 1950 to advise the government on scientific policy, and establish the policies of the NSF. The 24 members plus Director all appointed by the President and confirmed by the Senate, approve the budget and work of the NSF. The NSB sponsors national honorary awards such as the Vannevar Bush Award which is awarded to senior scientists for public service in science and technology.

The Society for Industrial and Applied Mathematics (SIAM) was incorporated in 1952 as a nonprofit organization to convey useful mathematical knowledge to computing and other professionals to solve practical, real-world problems. SIAM has over 13,000 members, and publishes 15 peer-reviewed journals – all electronic, and about 25 books each year. SIAM sponsors an annual MP3 high school contest with over $80,000 in scholarship prizes, funded by the Moody's Foundation.

Advanced Computing Systems Professional and Technical Association (USENIX) was created in 1975 by engineers, system administrators, scientists, and technicians to foster innovation, and exploration of the cutting edge of the computing world. USENIX hosts technical and system administration conferences, Informal, specific-topic conferences such as security, internet technology, and mobile computing, a tutorial program, a Special Interest Group for system administrators (SAGE), an online library, student programs, and participates in various standards efforts.

The Task Force on American Innovation is comprised of organizations from industry, professional groups, and academia. It advocates increased federal support for research in the physical sciences and engineering. Formed in 2004, the Task Force urges strong, sustained increases for research budgets at the NSF, DOE, Office of Science NIST, NASA, DOD. In 2010, corporate members include Agilent Technologies, Applied Materials, Google, IBM, Infineon, Intel, Microsoft, Northrop Grumman, P&G, Texas Instruments.

The Internet Society (ISOC) is a nonprofit organization founded in 1992 to address issues that confront the future of the Internet, and is the organizational home for the groups responsible for Internet infrastructure standards, including the Internet Engineering Task Force (IETF) and the Internet Architecture Board (IAB). With offices in Washington D.C., and Geneva, Switzerland, it is dedicated to ensuring the open development, evolution and use of the Internet for the benefit of people throughout the world.

Conclusion

The past 30 years has seen the production of over 20,000 Ph.D.s in computer science, and many master and bachelor degrees, and corporate certificates. The computing infrastructure created by these talented professionals has been of such high and lasting quality, that there have been predictions that fewer computer scientists will be needed in the future. On the other hand, the ability to gather and analyze huge data sets is changing the nature of scientific investigation. The traditional scientific method began with a hypothesis. The "modern scientific method" begins with data mining and a search for patterns. Computing has become part of ever more areas of human endeavor – from computational biology to computational vision. We have seen the number of computer science specialties grow from 9 in 2001, to 19 in 2008.

The United States has a depth of intellectual visionary and monitoring resource – world-class universities, the National Academies, 16 national laboratories, government and professional associations, and numerous military and corporate research units. In addition, it is one of the most philanthropic nations in the world, with individuals, foundations and companies gifting the promotion of computer science and mathematics.

Competition for students and jobs is serious. Asian countries are building vibrant, glamorous universities, but the best students still attend US institutions. The fastest trains in the world are not built in the US. Neither are the tallest buildings or the biggest companies. But, despite offshoring, foreign graduates still desire US residency and citizenship. The United States is known for bold imagination, and for empowering technological vitality, to which every one of the chapters in this book attests. Using Moore's Law as a metaphor, in theory and practice, the hardware and the software of computer science will continue to grow, and to explore the most imaginative and substantive of ideas.

Epilogue

The last Chap. 17, is a valuable catalog of real statistics of the "field", but it would be somewhat unsatisfactory and unfulfilling as an ending of what we hope is a stimulating report on the state of the Computer Science field. As editors, we believe it is appropriate, indeed incumbent on us, to end this book with some stimulating philosophical closing remarks addressed to our readers for their pleasure and edification. This brief epilogue may serve this purpose.

We have tried to present Computer Science as a Union of its hardware and software sides glued together by a theory at its heart. From the outset, as stated in the Introduction (Chap. 1), it was not our intent to provide complete coverage of the State of the Computer Science Union, and we did not do so. Computer Science, as you have hopefully learned from reading this book, is a multi-faceted dynamically evolving discipline. We have chosen to organize our report on its state by focusing on two of its main facets or sides: software and hardware (Chaps. 4 and 5). As we declared in the Introduction, we regard these two sides, as Turing did, not as independent entities but rather as two sides glued into a single interrelated whole, part of the glue being in a third constituent which we regard as the heart or essence of Computer Science. The heart (Chap. 3) includes such intrinsic properties as Church-Turing computability, *Unsolvability* and *Undecidability* (Turing's initial interest), which are properties inherent in the classical symbolic logic practiced by the brilliant mathematician Hilbert and the equally brilliant logicians Whitehead and Russell . (Read it in Appendix G of Chap. 3). It also includes the amazing complexity of solvable problems (Chaps. 12 and 13). The complexity of Turing computability has raised complexity issues about the world outside computation. How can a simple device like a Turing machine engender such complex behaviors? Are the world phenomena which machines model or simulate so complex? Or is the Church-Turing thesis itself at fault? Would some other thesis about computation yield less complex models? Do any of the book chapters provide hints at answers? What about quantum computers (Chap. 14) or fuzzy logic (Chap. 16)? These chapters will bear re-reading and investigation of their references.

Have we covered enough of the main infrastructure of Computer Science? We have covered many important subdisciplines that are active parts of Computer

E.K. Blum and A.V. Aho (eds.), *Computer Science: The Hardware, Software and Heart of It*, DOI 10.1007/978-1-4614-1168-0,

Science, such as Computer Networks (Chap. 7), Databases (Chap. 10) and Distributed Computing (Chaps. 8 and 9), although we may not have given sufficient attention to cloud computing in Chaps. 8 and 9. We have not explained the still unique Google search algorithm. The mathematics of graph theory (Chap. 7 Appendix) suggests that PageRank is the clue, but a reading of the original Brin-Page article which we did not cover will indicate how extensive is the Google computer search platform. We have also omitted coverage of the important subject known as artificial intelligence, since it is in many ways an independent discipline.

Somewhat perversely, we did cover the related topic of fuzzy logic, since we wish to dispel many misconstrued views of fuzzy logic as it impacts computer science. Another omission is our failure to cover the relatively new topic of computer games. On the other hand, we do cover the relatively new development of quantum computing (Chap. 14), which impacts many traditional issues in computer science, even possibly the properties which lie within its heart. Here, as did our author, we are careful to not make unfounded inferences. Were we too careful perhaps? Time will tell.

As for our style of presentation, we have tried diligently to fulfill our promise to cover topics equally on two levels, one intuitive and the other technical. A balanced exposition was not always possible. So, for example, the reader may have found that Chap. 15 on Numerical Thinking is heavily technical and mathematical, although it conveys an important intuitive message that penetrates the technical veil and readers will observe that it covers new aspects of traditional numerical analysis such as the fundamental problem of solving $Ax = b$, for large matrices A. This chapter will bear much re-reading.

In quite a different approach, the last Chap. 17, provides many useful statistics that shed light on the state of the Computer Science Union. These statistics could be a basis for predictions about the future of various practical aspects of Computer Science. Again, as we stated in the Introduction, we have resisted the temptation to be prophetic. The reader can perhaps infer the future condition of some parts of the state of the Union by extrapolating the statistics presented.

We trust that most readers have gained a better understanding of many of the recent developments in Computer Science. We also hope they have gained an appreciation of the exciting new impact of these developments on the daily lives of the world's citizens. In fact, one prediction we can safely make is that the impact of Computer Science on the lives of ordinary citizens of the planet earth will continue to widen and deepen in the coming decades.