Modern Heuristics

Contents

Articles

References

Article Licenses

Introduction

Heuristic

Heuristic (pronounced /hjʉˈrɪstɨk/, from the Greek "Εὑρίσκω" for "find" or "discover"): problem solving, learning and discovery. In Greek it is derived from the verb called *heuriskein* which means "to find". Archimedes is said to have shouted "*Heureka*" after discovering the principle of flotation in his bath, which is later converted to Eureka. A heuristic method is used to rapidly come to a solution that is hoped to be close to the best possible answer, or 'optimal solution'. A heuristic is a "rule of thumb", an educated guess, an intuitive judgment or simply common sense. A heuristic is a general way of solving a problem. *Heuristics* as a noun is another name for heuristic methods.

In more precise terms, heuristics stand control problem solving in human beings and machines.[1]

Example

It may be argued that the most fundamental heuristic is trial and error, which can be used in everything from matching bolts to bicycles to finding the values of variables in algebra problems.

Here are a few other commonly used heuristics, from Polya's 1945 book, *How to Solve It*:[2]

- If you are having difficulty understanding a problem, try drawing a picture.
- If you can't find a solution, try assuming that you have a solution and seeing what you can derive from that ("working backward").
- If the problem is abstract, try examining a concrete example.
- Try solving a more general problem first (the "inventor's paradox": the more ambitious plan may have more chances of success).

Psychology

In psychology, heuristics are simple, efficient rules, hard-coded by evolutionary processes or learned, which have been proposed to explain how people make decisions, come to judgments, and solve problems, typically when facing complex problems or incomplete information. These rules work well under most circumstances, but in certain cases lead to systematic errors or cognitive biases.

Although much of the work of discovering heuristics in human decision-makers has been done by Amos Tversky and Daniel Kahneman[3] , the concept was originally introduced by Nobel laureate Herbert Simon. Gerd Gigerenzer focuses on how heuristics can be used to make judgments that are in principle accurate, rather than producing cognitive biases – heuristics that are "fast and frugal".[4]

In 2002, Daniel Kahneman and Shane Frederick proposed that cognitive heuristics work by a process called *attribute substitution* which happens without conscious awareness.[5] According to this theory, when someone makes a judgment (of a *target attribute*) that is computationally complex, they instead substitute a more easily calculated *heuristic attribute*. In effect, they deal with a cognitively difficult problem by answering a more simple problem, without being aware that this is happening.[5] This theory explains cases where judgments fail to show regression toward the mean.[6]

Theorized psychological heuristics

Well known

- Anchoring and adjustment
- Availability heuristic
- Representativeness heuristic
- Naïve diversification
- Escalation of commitment

Less well known

- Affect heuristic
- Contagion heuristic
- Effort heuristic
- Familiarity heuristic
- Fluency heuristic
- Gaze heuristic
- Peak-end rule
- Recognition heuristic
- Scarcity heuristic
- Similarity heuristic
- Simulation heuristic
- Social proof
- Take-the-best heuristic

Philosophy

In philosophy, especially in Continental European philosophy, the adjective "heuristic" (or the designation "heuristic device") is used when an entity X exists to enable understanding of, or knowledge concerning, some other entity Y. A good example is a model which, as it is never identical with what it models, is a heuristic device to enable understanding of what it models. Stories, metaphors, etc., can also be termed heuristic in that sense. A classic example is the notion of utopia as described in Plato's best-known work, *The Republic*. This means that the "ideal city" as depicted in *The Republic* is not given as something to be pursued, or to present an orientation-point for development; rather, it shows how things would have to be connected, and how one thing would lead to another (often with highly problematic results), if one would opt for certain principles and carry them through rigorously.

"Heuristic" is also often commonly used as a noun to describe a rule-of-thumb, procedure, or method.[7] Philosophers of science have emphasized the importance of heuristics in creative thought and constructing scientific theories.[8] (See the logic of discovery, and philosophers such as Imre Lakatos,[9] Lindley Darden, and others.)

Law

In legal theory, especially in the theory of law and economics, heuristics are used in the law when case-by-case analysis would be impractical, insofar as "practicality" is defined by the interests of a governing body.[10]

For instance, in many states in the United States the legal drinking age is 21, because it is argued that people need to be mature enough to make decisions involving the risks of alcohol consumption. However, assuming people mature at different rates, the specific age of 21 would be too late for some and too early for others. In this case, the somewhat arbitrary deadline is used because it is impossible or impractical to tell whether one individual is mature enough that society can trust them with that kind of responsibility. Some proposed changes, however, have included the completion of an alcohol education course rather than the attainment of 21 years of age as the criterion for legal alcohol possession. This would situate youth alcohol policy more on a case-by-case model and less on a heuristic one, since the completion of such a course would presumably be voluntary and not uniform across the population.

The same reasoning applies to patent law. Patents are justified on the grounds that inventors need to be protected in order to have incentive to invent. It is therefore argued that, in society's best interest, inventors should be issued with a temporary government-granted monopoly on their product, so that they can recoup their investment costs and make economic profit for a limited period of time. In the United States the length of this temporary monopoly is 20 years from the date the application for patent was filed, though the monopoly does not actually begin until the application

has matured into a patent. However, like the drinking-age problem above, the specific length of time would need to be different for every product in order to be efficient; a 20-year term is used because it is difficult to tell what the number should be for any individual patent. More recently, some, including University of North Dakota law professor Eric E. Johnson, have argued that patents in different kinds of industries – such as software patents – should be protected for different lengths of time.[11]

Computer science

In computer science, a heuristic is a technique designed to solve a problem that ignores whether the solution can be proven to be correct, but which usually produces a good solution or solves a simpler problem that contains or intersects with the solution of the more complex problem. Most real-time, and even some on-demand, anti-virus scanners use heuristic signatures to look for specific attributes and characteristics for detecting viruses and other forms of malware.

Heuristics are intended to gain computational performance or conceptual simplicity, potentially at the cost of accuracy or precision.

In their Turing Award acceptance speech, Herbert Simon and Allen Newell discuss the Heuristic Search Hypothesis: a physical symbol system will repeatedly generate and modify known symbol structures until the created structure matches the solution structure.

That is, each successive iteration depends upon the step before it, thus the heuristic search learns what avenues to pursue and which ones to disregard by measuring how close the current iteration is to the solution. Therefore, some possibilities will never be generated as they are measured to be less likely to complete the solution.

A heuristic method can accomplish its task by using search trees. However, instead of generating all possible solution branches, a heuristic selects branches more likely to produce outcomes than other branches. It is selective at each decision point; picking branches that are more likely to produce solutions. [12]

Human-computer interaction

In human-computer interaction, heuristic evaluation is a usability-testing technique devised by expert usability consultants. In heuristic evaluation, the user interface is reviewed by experts and its compliance to *usability heuristics* (broadly stated characteristics of a good user interface, based on prior experience) is assessed, and any violating aspects are recorded.

Heuristic Considerations in the Design of Software Applications

A well-designed user interface enables users to intuitively navigate complex systems, without difficulty. It guides the user when necessary using tooltips, help buttons, invitations to chat with support, etc., providing help when needed. This is not always an easy process though.

Software developers and the targeted end-users alike each disregard heuristics at their own peril. End users often need to increase their understanding of the basic framework that a project entails (so that their expectations are realistic), and developers often need to push to learn more about their target audience (so that their learning styles can be judged somewhat). Business rules crucial to the organization are often so obvious to the end-user that they are not conveyed to the developer, who may be unaquainted with the particular field of endeavor the application is meant to serve.

A proper Software Requirement Specification (SRS) models the heuristics of how a user will process the information being rendered on-screen. An SRS is ideally shared with the end-user well before the actual Software Design Specification (SDS) is written and the application is developed, so that the user's feedback about their experience can be used to adapt the design of the application. This saves much time in the Software Development Life Cycle

(SDLC). Unless heurisitics are considered adequately enough, the project will likely suffer many implementation problems and setbacks.

Engineering

In engineering, a heuristic is an experience-based method that can be used as an aid to solve process design problems, varying from size of equipment to operating conditions. By using heuristics, time can be reduced when solving problems. There are several methods which are available to engineers. These include Failure mode and effects analysis and Fault tree analysis. The former relies on a group of qualified engineers to evaluate problems, rank them in order of importance and then recommend solutions. The methods of forensic engineering are an important source of information for investigating problems, especially by elimination of unlikely causes and using the weakest link principle.

Because heuristics are fallible, it is important to understand their limitations. They are intended to be used as aids in order to make quick estimates and preliminary process designs.

Pitfalls of heuristics

Heuristic algorithms are often employed because they may be seen to "work" without having been mathematically proven to meet a given set of requirements.

Great care must be given when employing a heuristic to solve a problem. One common pitfall in implementing a heuristic method to meet a requirement comes when the engineer or designer fails to realize that the current data set does not necessarily represent future system states.

While the existing data can be pored over and an algorithm can be devised to successfully handle the current data, it is imperative to ensure that the heuristic method employed is capable of handling future data sets. This means that the engineer or designer must fully understand the rules that generate the data and develop the algorithm to meet those requirements and not just address the current data sets.

Statistical analysis should be conducted when employing heuristics to estimate the probability of incorrect outcomes.

See also

- Behavioral economics – an economic subfield with heuristics as one of its main arguments
  - Daniel Kahneman - psychologist who won the 2002 Nobel Prize in Economics for his work with heuristics and human beings
- Algorithm
- Failure mode and effects analysis
- Problem solving
- Teachable moment
- List of cognitive biases

Further reading

- *How To Solve It: Modern Heuristics*, Zbigniew Michalewicz and David B. Fogel, Springer Verlag, 2000. ISBN 3-540-66061-5
- Russell, Stuart J.; Norvig, Peter (2003), *Artificial Intelligence: A Modern Approach* [13] (2nd ed.), Upper Saddle River, New Jersey: Prentice Hall, ISBN 0-13-790395-2
- The Problem of Thinking Too Much [14], 2002–12–11, Persi Diaconis

External links

- The Heuristic Wiki [15]
- Heuristics and artificial intelligence in finance and investment [16] – The use of heuristics and AI techniques in finance and investment.
- "Discovering Assumptions" [17] by Paul Niquette — Highly recommended

References

[1] Pearl, Judea (1983). *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. New York, Addison-Wesley, p. vii.

[2] Polya, George (1945) *How To Solve It: A New Aspect of Mathematical Method*, Princeton, NJ: Princeton University Press. ISBN 0-691-02356-5 ISBN 0-691-08097-6

[3] Daniel Kahneman, Amos Tversky and Paul Slovic, eds. (1982) *Judgment under Uncertainty: Heuristics & Biases*. Cambridge, UK, Cambridge University Press ISBN 0-521-28414-7

[4] Gerd Gigerenzer, Peter M. Todd, and the ABC Research Group (1999). *Simple Heuristics That Make Us Smart*. Oxford, UK, Oxford University Press. ISBN 0-19-514381-7

[5] Kahneman, Daniel; Shane Frederick (2002). "Representativeness Revisited: Attribute Substitution in Intuitive Judgment". in Thomas Gilovich, Dale Griffin, Daniel Kahneman. *Heuristics and Biases: The Psychology of Intuitive Judgment*. Cambridge: Cambridge University Press. pp. 49–81. ISBN 9780521796798. OCLC 47364085.

[6] Kahneman, Daniel (December 2003). "Maps of Bounded Rationality: Psychology for Behavioral Economics". *American Economic Review* (American Economic Association) **93** (5): 1449–1475. doi:10.1257/000282803322655392. ISSN 0002-8282.

[7] K. M. Jaszczolt (2006). "Defaults in Semantics and Pragmatics" (http://plato.stanford.edu/entries/defaults-semantics-pragmatics/), The Stanford Encyclopedia of Philosophy, ISSN 1095-5054

[8] Roman Frigg and Stephan Hartmann (2006). "Models in Science" (http://plato.stanford.edu/entries/models-science/), The Stanford Encyclopedia of Philosophy, ISSN 1095-5054

[9] Olga Kiss (2006). "Heuristic, Methodology or Logic of Discovery? Lakatos on Patterns of Thinking" (http://www.mitpressjournals.org/doi/pdf/10.1162/posc.2006.14.3.302), Perspectives on Science, vol. 14, no. 3, pp. 302-317, ISSN 1063-6145

[10] Gerd Gigerenzer and Christoph Engel, eds. (2007). *Heuristics and the Law*, Cambridge, The MIT Press, ISBN 978-0-262-07275-5

[11] Eric E. Johnson (2006). "Calibrating Patent Lifetimes" (http://www.eejlaw.com/writings/Johnson_Calibrating_Patent_Lifetimes.pdf), Santa Clara Computer & High Technology Law Journal, vol. 22, p. 269-314

[12] Newell, A. & Simon, H. A. (1976). Computer science as empirical inquiry: symbols and search. Comm. Of the ACM. 19, 113-126.

[13] http://aima.cs.berkeley.edu/

[14] http://www-stat.stanford.edu/~cgates/PERSI/papers/thinking.pdf

[15] http://greenlightwiki.com/heuristic

[16] http://www.oocities.com/francorbusetti/index.html

[17] http://niquette.com/books/sophmag/heurist.htm

Combinatorial optimization

Combinatorial optimization is a branch of optimization. Its domain is optimization problems where the set of feasible solutions is discrete or can be reduced to a discrete one, and the goal is to find the best possible solution.

It is a branch of applied mathematics and computer science, related to operations research, algorithm theory and computational complexity theory that sits at the intersection of several fields, including artificial intelligence, mathematics and software engineering.

Some research literature[1] considers discrete optimization to consist of integer programming together with combinatorial optimization (which in turn is composed of optimization problems dealing with graphs, matroids, and related structures) although all of these topics have closely intertwined research literature.

Example problems

- Vehicle routing problem
- Traveling salesman problem (TSP)
- Minimum spanning tree problem
- Linear programming (if the solution space is the choice of which variables to make basic)
- Integer programming
- Eight queens puzzle (A constraint satisfaction problem. When applying standard combinatorial optimization algorithms to this problem, one would usually treat the goal function as the number of unsatisfied constraints (say number of attacks) rather than as a single boolean indicating whether the whole problem is satisfied or not.)
- Knapsack problem
- Cutting stock problem

Methods

There is a large amount of literature on polynomial-time algorithms for certain special classes of discrete optimization, a considerable amount of it unified by the theory of linear programming. Some examples of combinatorial optimization problems that fall into this framework are shortest paths and shortest path trees, flows and circulations, spanning trees, matching, and matroid problems.

For NP-complete discrete optimization problems, current research literature includes the following topics:

- polynomial-time exactly-solvable special cases of the problem at hand (e.g. see fixed-parameter tractable)
- algorithms that perform well on "random" instances (e.g. for TSP)
- approximation algorithms that run in polynomial time and find a solution that is "close" to optimal
- solving real-world instances that arise in practice and do not necessarily exhibit the worst-case behaviour inherent in NP-complete problems (e.g. TSP instances with tens of thousands of nodes[2]).

Combinatorial optimization problems can be viewed as searching for the best element of some set of discrete items, therefore, in principle, any sort of search algorithm or metaheuristic can be used to solve them. However, generic search algorithms are not guaranteed to find an optimal solution, nor are they guaranteed to run quickly (in polynomial time). (Since some discrete optimization problems are NP-complete, e.g. the travelling salesman problem, this is expected unless P=NP.)

References

[1] "Discrete Optimization" (http://www.elsevier.com/locate/disopt). Elsevier. . Retrieved 2009-06-08.
[2] Bill Cook. "Optimal TSP Tours" (http://www.tsp.gatech.edu/optimal/index.html). . Retrieved 2009-06-08.

- Alexander Schrijver; *A Course in Combinatorial Optimization* (http://homepages.cwi.nl/~lex/files/dict.pdf) February 1, 2006 (© A. Schrijver)
- William J. Cook, William H. Cunningham, William R. Pulleyblank, Alexander Schrijver; *Combinatorial Optimization*; John Wiley & Sons; 1 edition (November 12, 1997); ISBN 0-471-55894-X.
- Jon Lee; "A First Course in Combinatorial Optimization"; Cambridge University Press; 2004; ISBN 0-521-01012-8.
- Pierluigi Crescenzi, Viggo Kann, Magnús Halldórsson, Marek Karpinski, Gerhard Woeginger, *A Compendium of NP Optimization Problems* (http://www.nada.kth.se/~viggo/wwwcompendium/).
- Christos H. Papadimitriou and Kenneth Steiglitz *Combinatorial Optimization : Algorithms and Complexity*; Dover Pubns; (paperback, Unabridged edition, July 1998) ISBN 0-486-40258-4.
- Arnab Das and Bikas K. Chakrabarti (Eds.) *Quantum Annealing and Related Optimization Methods*, Lecture Note in Physics, Vol. **679**, Springer, Heidelberg (2005)
- Journal of Combinatorial Optimization (http://www.kluweronline.com/issn/1382-6905)

External links

- Integer programming (http://people.brunel.ac.uk/~mastjjb/jeb/or/ip.html) notes, J E Beasley.
- Java Combinatorial Optimization Platform (http://sourceforge.net/projects/jcop/) open source project.

Slack variable

In linear programming, a **slack variable** is a variable that is added to a constraint to turn the inequality into an equation. This is required to turn an inequality into an equality where a linear combination of variables is less than or equal to a given constant in the former. As with the other variables in the augmented constraints, the slack variable cannot take on negative values, as the Simplex algorithm requires them to be positive or zero.

- If the slack variable associated with a constraint is *zero* in a given state, the constraint is **binding**, as the constraint restricts the possible changes of the point.
- If a slack variable is *positive* in a given state, the constraint is **non-binding**, as the constraint does not restrict the possible changes of the point.
- If a slack variable is *negative* in a given state, the point is **infeasible**, and not allowed, as it does not satisfy the constraint.

Example

By introducing the slack variable $\mathbf{y} \geq 0$, the inequality $\mathbf{Ax} \leq \mathbf{b}$ can be converted to the equation $\mathbf{Ax} + \mathbf{y} = \mathbf{b}$.

Embedding in orthant

Slack variables give an embedding of a polytope $P \hookrightarrow (\mathbf{R}_{\geq 0})^f$ into the standard *f*-orthant, where *f* is the number of constraints (facets of the polytope). This map is one-to-one (slack variables are uniquely determined) but not onto (not all combinations can be realized), and is expressed in terms of the *constraints* (linear functionals, covectors).
Slack variables are *dual* to generalized barycentric coordinates, and, dually to generalized barycentric coordinates (which are not unique but can all be realized), are uniquely determined, but cannot all be realized.

Dually, generalized barycentric coordinates express a polytope with *n* vertices (dual to facets), regardless of dimension, as the *image* of the standard $(n-1)$-simplex, which has *n* vertices – the map is onto: $\Delta^{n-1} \twoheadrightarrow P$, and expresses points in terms of the *vertices* (points, vectors). The map is one-to-one if and only if the polytope is a simplex, in which case the map is an isomorphism; this corresponds to a point not having *unique* generalized barycentric coordinates.

See also

- Surplus variable
- Simplex algorithm

External links

- An explanation of the Simplex method [1]

References

[1] http://www-fp.mcs.anl.gov/otc/Guide/CaseStudies/simplex/standard.html

Constraint (information theory)

Constraint in information theory refers to the degree of statistical dependence between or among variables.

Garner[1] provides a thorough discussion of various forms of constraint (internal constraint, external constraint, total constraint) with application to pattern recognition and psychology.

See also

- Mutual Information
- Total Correlation
- Interaction information

References

[1] Garner W R (1962). *Uncertainty and Structure as Psychological Concepts*, John Wiley & Sons, New York.

Problem Representations

Problem

A **problem** is an obstacle which makes it difficult to achieve a desired goal, objective or purpose. It refers to a situation, condition, or issue that is yet unresolved. In a broad sense, a problem exists when an individual becomes aware of a significant difference between what actually is and what is desired.

Problem solving

Every theoretical problem asks for an answer or solution. Trying to find a solution to a problem is known as problem solving. That is, a problem is a gap between an actual and desired situation. The time it takes to solve a problem is a way of measuring complexity.[1] Many problems have no discovered solution and are therefore classified as an open problem.

From the mid 20th century, the field of theoretical computer science has explored the use of computers to solve problems.

Examples

- Mathematical problem is a question about mathematical objects and structures that may require a distinct answer or explanation or proof. Examples include word problems at school level or deeper problems such as shading a map with only four colours.
- In society, a problem can refer to particular social issues, which if solved would yield social benefits, such as increased harmony or productivity, and conversely diminished hostility and disruption.
- In business and engineering, a problem is a difference between actual conditions and those that are required or desired. Often, the causes of a problem are not known, in which case root cause analysis is employed to find the causes and identify corrective actions.
- In chess, a problem is a puzzle set by somebody using chess pieces on a chess board, for others to get instruction or intellectual satisfaction from determining the solution.
- In theology, there is what is referred to as the Synoptic Problem, which includes in its discourse a concern for assumptions of historical accuracy that are challenged by apparent contradictions in the Gospels' accounts of allegedly historical events.
- In academic discourse a problem is a challenge to an assumption, an apparent conflict that requires synthesis and reconciliation. It is a normal part of systematic thinking, the address of which adds to or detracts from the veracity of a conclusion or idea.
- An optimization problem is finding the best solution from all feasible solutions. A good example of this type of problem is the travelling salesperson problem which is based on calculating the most efficient route between many places
- In computability theory a decision problem requires a simple yes-or-no answer.
- In rock climbing a problem is a series of rocks that forces the climber to climb.
- In reading, a problem is a combination of a series of words with the overall plotline, which the reader must attempt to decipher.
- In walking, a mobility problem is presented. Motion is achieved via mechanical interaction of the legs and a surface.

See also

- Challenge
- Heuristic
- How to Solve It
- Question
- Problem-based learning
- Problematization
- Worry

References

[1] *The Puzzle Master*. Alexandria, Virginia, USA: Time-Life Books. 1989. pp. 32. ISBN 0809709287.

Problem solving

Problem solving is a mental process and is part of the larger problem process that includes problem finding and problem shaping. Considered the most complex of all intellectual functions, problem solving has been defined as higher-order cognitive process that requires the modulation and control of more routine or fundamental skills. [1] Problem solving occurs when an organism or an artificial intelligence system needs to move from a given state to a desired goal state.

Overview

The nature of human problem solving methods has been studied by psychologists over the past hundred years. There are several methods of studying problem solving, including; introspection, behaviorism, simulation, computer modeling and experiment.

Beginning with the early experimental work of the Gestaltists in Germany (e.g. Duncker, 1935 [2]), and continuing through the 1960s and early 1970s, research on problem solving typically conducted relatively simple, laboratory tasks (e.g. Duncker's "X-ray" problem; Ewert & Lambert's 1932 "disk" problem, later known as Tower of Hanoi) that appeared novel to participants (e.g. Mayer, 1992 [3]). Various reasons account for the choice of simple novel tasks: they had clearly defined optimal solutions, they were solvable within a relatively short time frame, researchers could trace participants' problem-solving steps, and so on. The researchers made the underlying assumption, of course, that simple tasks such as the Tower of Hanoi captured the main properties of "real world" problems, and that the cognitive processes underlying participants' attempts to solve simple problems were representative of the processes engaged in when solving "real world" problems. Thus researchers used simple problems for reasons of convenience, and thought generalizations to more complex problems would become possible. Perhaps the best-known and most impressive example of this line of research remains the work by Allen Newell and Herbert Simon [4] .

Simple laboratory-based tasks can be useful in explicating the steps of logic and reasoning that underlie problem solving; however, they omit the complexity and emotional valence of "real-world" problems. In clinical psychology, researchers have focused on the role of emotions in problem solving (D'Zurilla & Goldfried, 1971; D'Zurilla & Nezu, 1982), demonstrating that poor emotional control can disrupt focus on the target task and impede problem resolution (Rath, Langenbahn, Simon, Sherr, & Diller, 2004). In this conceptualization, human problem solving consists of two related processes: problem orientation, the motivational/attitudinal/affective approach to problematic situations and problem-solving skills, the actual cognitive-behavioral steps, which, if successfully implemented, lead to effective problem resolution. Working with individuals with frontal lobe injuries, neuropsychologists have discovered that deficits in emotional control and reasoning can be remediated, improving the capacity of injured

persons to resolve everyday problems successfully (Rath, Simon, Langenbahn, Sherr, & Diller, 2003).

Europe

In Europe, two main approaches have surfaced, one initiated by Donald Broadbent (1977; see Berry & Broadbent, 1995) in the United Kingdom and the other one by Dietrich Dörner (1975, 1985; see Dörner & Wearing, 1995) in Germany. The two approaches have in common an emphasis on relatively complex, semantically rich, computerized laboratory tasks, constructed to resemble real-life problems. The approaches differ somewhat in their theoretical goals and methodology, however. The tradition initiated by Broadbent emphasizes the distinction between cognitive problem-solving processes that operate under awareness versus outside of awareness, and typically employs mathematically well-defined computerized systems. The tradition initiated by Dörner, on the other hand, has an interest in the interplay of the cognitive, motivational, and social components of problem solving, and utilizes very complex computerized scenarios that contain up to 2,000 highly interconnected variables (e.g., Dörner, Kreuzig, Reither & Stäudel's 1983 LOHHAUSEN project; Ringelband, Misiak & Kluwe, 1990). Buchner (1995) describes the two traditions in detail.

To sum up, researchers' realization that problem-solving processes differ across knowledge domains and across levels of expertise (e.g. Sternberg, 1995) and that, consequently, findings obtained in the laboratory cannot necessarily generalize to problem-solving situations outside the laboratory, has during the past two decades led to an emphasis on real-world problem solving. This emphasis has been expressed quite differently in North America and Europe, however. Whereas North American research has typically concentrated on studying problem solving in separate, natural knowledge domains, much of the European research has focused on novel, complex problems, and has been performed with computerized scenarios (see Funke, 1991, for an overview).

USA and Canada

In North America, initiated by the work of Herbert Simon on learning by doing in semantically rich domains (e.g. Anzai & Simon, 1979; Bhaskar & Simon, 1977), researchers began to investigate problem solving separately in different natural knowledge domains – such as physics, writing, or chess playing – thus relinquishing their attempts to extract a global theory of problem solving (e.g. Sternberg & Frensch, 1991). Instead, these researchers have frequently focused on the development of problem solving within a certain domain, that is on the development of expertise (e.g. Anderson, Boyle & Reiser, 1985; Chase & Simon, 1973; Chi, Feltovich & Glaser, 1981).

Areas that have attracted rather intensive attention in North America include such diverse fields as:

- Reading (Stanovich & Cunningham, 1991)
- Writing (Bryson, Bereiter, Scardamalia & Joram, 1991)
- Calculation (Sokol & McCloskey, 1991)
- Political decision making (Voss, Wolfe, Lawrence & Engle, 1991)
- Managerial problem solving (Wagner, 1991)
- Lawyers' reasoning (Amsel, Langer & Loutzenhiser, 1991)
- Mechanical problem solving (Hegarty, 1991)
- Problem solving in electronics (Lesgold & Lajoie, 1991)
- Computer skills (Kay, 1991)
- Game playing (Frensch & Sternberg, 1991)
- Personal problem solving (Heppner & Krauskopf, 1987)
- Mathematical problem solving (Polya, 1945; Schoenfeld, 1985)
- Social problem solving (D'Zurilla & Goldfreid, 1971; D'Zurilla & Nezu, 1982)
- Problem solving for innovations and inventions: TRIZ (Altshuller, 1973, 1984, 1994)

Characteristics of difficult problems

As elucidated by Dietrich Dörner and later expanded upon by Joachim Funke, difficult problems have some typical characteristics that can be summarized as follows:

- Intransparency (lack of clarity of the situation)
  - commencement opacity
  - continuation opacity
- Polytely (multiple goals)
  - inexpressiveness
  - opposition
  - transience
- Complexity (large numbers of items, interrelations and decisions)
  - enumerability
  - connectivity (hierarchy relation, communication relation, allocation relation)
  - heterogeneity
- Dynamics (time considerations)
  - temporal constraints
  - temporal sensitivity
  - phase effects
  - dynamic unpredictability

The resolution of difficult problems requires a direct attack on each of these characteristics that are encountered.

In reform mathematics, greater emphasis is placed on problem solving relative to basic skills, where basic operations can be done with calculators. However some "problems" may actually have standard solutions taught in higher grades. For example, kindergarteners could be asked how many fingers are there on all the gloves of 3 children, which can be solved with multiplication. [5]

Problem-solving techniques

- Abstraction: solving the problem in a model of the system before applying it to the real system.
- Analogy: using a solution that solved an analogous problem.
- Brainstorming: (especially among groups of people) suggesting a large number of solutions or ideas and combining and developing them until an optimum is found.
- Divide and conquer: breaking down a large, complex problem into smaller, solvable problems.
- Hypothesis testing: assuming a possible explanation to the problem and trying to prove (or, in some contexts, disprove) the assumption.
- Lateral thinking: approaching solutions indirectly and creatively.
- Means-ends analysis: choosing an action at each step to move closer to the goal.
- Method of focal objects: synthesizing seemingly non-matching characteristics of different objects into something new.
- Morphological analysis: assessing the output and interactions of an entire system.
- Reduction: transforming the problem into another problem for which solutions exist.
- Research: employing existing ideas or adapting existing solutions to similar problems.
- Root cause analysis: eliminating the cause of the problem.
- Trial-and-error: testing possible solutions until the right one is found.

Problem-solving methodologies

- Eight Disciplines Problem Solving
- GROW model
- *How to solve it*
- Kepner-Tregoe
- Southbeach Notation
- PDCA
- RPR Problem Diagnosis
- TRIZ (Teoriya Resheniya Izobretatelskikh Zadatch, "theory of solving inventor's problems")

Example applications

Problem solving is of crucial importance in engineering when products or processes fail, so corrective action can be taken to prevent further failures. Perhaps of more value, problem solving can be applied to a product or process prior to an actual fail event ie. a potential problem can be predicted, analyzed and mitigation applied so the problem never actually occurs. Techniques like Failure Mode Effects Analysis can be used to proactively reduce the likelihood of problems occurring. Forensic engineering is an important technique of failure analysis which involves tracing product defects and flaws. Corrective action can then be taken to prevent further failures.

See also

- Artificial intelligence
- C-K Design Theory
- Creative problem solving
- Divergent thinking
- Educational psychology
- Executive function
- Forensic engineering
- Heuristics
- Innovation
- Intelligence amplification
- Inquiry
- Logical reasoning
- Problem statement
- Herbert Simon
- Thought
- Transdisciplinary studies
- Troubleshooting
- Wicked problem

References

- Amsel, E., Langer, R., & Loutzenhiser, L. (1991). Do lawyers reason differently from psychologists? A comparative design for studying expertise. In R. J. Sternberg & P. A. Frensch (Eds.), *Complex problem solving: Principles and mechanisms* (pp. 223-250). Hillsdale, NJ: Lawrence Erlbaum Associates. ISBN 978-0-8058-1783-6
- Anderson, J. R., Boyle, C. B., & Reiser, B. J. (1985). "Intelligent tutoring systems". *Science* **228** (4698): 456–462. doi:10.1126/science.228.4698.456. PMID 17746875.
- Anzai, K., & Simon, H. A. (1979) (1979). "The theory of learning by doing". *Psychological Review* **86** (2): 124–140. doi:10.1037/0033-295X.86.2.124. PMID 493441.
- Beckmann, J. F., & Guthke, J. (1995). Complex problem solving, intelligence, and learning ability. In P. A. Frensch & J. Funke (Eds.), *Complex problem solving: The European Perspective* (pp. 177-200). Hillsdale, NJ: Lawrence Erlbaum Associates.
- Berry, D. C., & Broadbent, D. E. (1995). Implicit learning in the control of complex systems: A reconsideration of some of the earlier claims. In P.A. Frensch & J. Funke (Eds.), *Complex problem solving: The European Perspective* (pp. 131-150). Hillsdale, NJ: Lawrence Erlbaum Associates.
- Bhaskar, R., & Simon, H. A. (1977). Problem solving in semantically rich domains: An example from engineering thermodynamics. *Cognitive Science*, 1, 193-215.
- Brehmer, B. (1995). Feedback delays in dynamic decision making. In P. A. Frensch & J. Funke (Eds.), *Complex problem solving: The European Perspective* (pp. 103-130). Hillsdale, NJ: Lawrence Erlbaum Associates.
- Brehmer, B., & Dörner, D. (1993). Experiments with computer-simulated microworlds: Escaping both the narrow straits of the laboratory and the deep blue sea of the field study. *Computers in Human Behavior*, 9, 171-184.
- Broadbent, D. E. (1977). Levels, hierarchies, and the locus of control. *Quarterly Journal of Experimental Psychology*, 29, 181-201.
- Bryson, M., Bereiter, C., Scardamalia, M., & Joram, E. (1991). Going beyond the problem as given: Problem solving in expert and novice writers. In R. J. Sternberg & P. A. Frensch (Eds.), *Complex problem solving: Principles and mechanisms* (pp. 61-84). Hillsdale, NJ: Lawrence Erlbaum Associates.
- Buchner, A. (1995). Theories of complex problem solving. In P. A. Frensch & J. Funke (Eds.), *Complex problem solving: The European Perspective* (pp. 27-63). Hillsdale, NJ: Lawrence Erlbaum Associates.
- Buchner, A., Funke, J., & Berry, D. C. (1995). Negative correlations between control performance and verbalizable knowledge: Indicators for implicit learning in process control tasks? *Quarterly Journal of Experimental Psychology*, 48A, 166-187.
- Chase, W. G., & Simon, H. A. (1973). Perception in chess. *Cognitive Psychology*, 4, 55-81.
- Chi, M. T. H., Feltovich, P. J., & Glaser, R. (1981). "Categorization and representation of physics problems by experts and novices" [6]. *Cognitive Science* **5**: 121–152. doi:10.1207/s15516709cog0502_2.
- Dörner, D. (1975). Wie Menschen eine Welt verbessern wollten [How people wanted to improve the world]. *Bild der Wissenschaft*, 12, 48-53.
- Dörner, D. (1985). Verhalten, Denken und Emotionen [Behavior, thinking, and emotions]. In L. H. Eckensberger & E. D. Lantermann (Eds.), *Emotion und Reflexivität* (pp. 157-181). München, Germany: Urban & Schwarzenberg.
- Dörner, D. (1992). Über die Philosophie der Verwendung von Mikrowelten oder "Computerszenarios" in der psychologischen Forschung [On the proper use of microworlds or "computer scenarios" in psychological research]. In H. Gundlach (Ed.), *Psychologische Forschung und Methode: Das Versprechen des Experiments. Festschrift für Werner Traxel* (pp. 53-87). Passau, Germany: Passavia-Universitäts-Verlag.
- Dörner, D., Kreuzig, H. W., Reither, F., & Stäudel, T. (Eds.). (1983). *Lohhausen. Vom Umgang mit Unbestimmtheit und Komplexität* [Lohhausen. On dealing with uncertainty and complexity]. Bern, Switzerland: Hans Huber.

- Dörner, D., & Wearing, A. (1995). Complex problem solving: Toward a (computer-simulated) theory. In P. A. Frensch & J. Funke (Eds.), *Complex problem solving: The European Perspective* (pp. 65-99). Hillsdale, NJ: Lawrence Erlbaum Associates.
- Duncker, K. (1935). *Zur Psychologie des produktiven Denkens* [The psychology of productive thinking]. Berlin: Julius Springer.
- Ewert, P. H., & Lambert, J. F. (1932). Part II: The effect of verbal instructions upon the formation of a concept. *Journal of General Psychology*, 6, 400-411.
- Eyferth, K., Schömann, M., & Widowski, D. (1986). Der Umgang von Psychologen mit Komplexität [On how psychologists deal with complexity]. *Sprache & Kognition*, 5, 11-26.
- Frensch, P. A., & Funke, J. (Eds.). (1995). *Complex problem solving: The European Perspective*. Hillsdale, NJ: Lawrence Erlbaum Associates.
- Frensch, P. A., & Sternberg, R. J. (1991). Skill-related differences in game playing. In R. J. Sternberg & P. A. Frensch (Eds.), *Complex problem solving: Principles and mechanisms* (pp. 343-381). Hillsdale, NJ: Lawrence Erlbaum Associates.
- Funke, J. (1991). Solving complex problems: Human identification and control of complex systems. In R. J. Sternberg & P. A. Frensch (Eds.), *Complex problem solving: Principles and mechanisms* (pp. 185-222). Hillsdale, NJ: Lawrence Erlbaum Associates.
- Funke, J. (1993). Microworlds based on linear equation systems: A new approach to complex problem solving and experimental results. In G. Strube & K.-F. Wender (Eds.), *The cognitive psychology of knowledge* (pp. 313-330). Amsterdam: Elsevier Science Publishers.
- Funke, J. (1995). Experimental research on complex problem solving. In P. A. Frensch & J. Funke (Eds.), *Complex problem solving: The European Perspective* (pp. 243-268). Hillsdale, NJ: Lawrence Erlbaum Associates.
- Funke, U. (1995). Complex problem solving in personnel selection and training. In P. A. Frensch & J. Funke (Eds.), *Complex problem solving: The European Perspective* (pp. 219-240). Hillsdale, NJ: Lawrence Erlbaum Associates.
- Goldstein F. C., & Levin H. S. (1987). Disorders of reasoning and problem-solving ability. In M. Meier, A. Benton, & L. Diller (Eds.), *Neuropsychological rehabilitation*. London: Taylor & Francis Group.
- Groner, M., Groner, R., & Bischof, W. F. (1983). Approaches to heuristics: A historical review. In R. Groner, M. Groner, & W. F. Bischof (Eds.), *Methods of heuristics* (pp. 1-18). Hillsdale, NJ: Lawrence Erlbaum Associates.
- Halpern, Diane F. (2002).Thought & Knowledge. Lawrence Erlbaum Associates. Worldcat Library Catalog [7]
- Hayes, J. (1980). *The complete problem solver*. Philadelphia: The Franklin Institute Press.
- Hegarty, M. (1991). Knowledge and processes in mechanical problem solving. In R. J. Sternberg & P. A. Frensch (Eds.), *Complex problem solving: Principles and mechanisms* (pp. 253-285). Hillsdale, NJ: Lawrence Erlbaum Associates.
- Heppner, P. P., & Krauskopf, C. J. (1987). An information-processing approach to personal problem solving. *The Counseling Psychologist*, 15, 371-447.
- Huber, O. (1995). Complex problem solving as multi stage decision making. In P. A. Frensch & J. Funke (Eds.), *Complex problem solving: The European Perspective* (pp. 151-173). Hillsdale, NJ: Lawrence Erlbaum Associates.
- Hübner, R. (1989). Methoden zur Analyse und Konstruktion von Aufgaben zur kognitiven Steuerung dynamischer Systeme [Methods for the analysis and construction of dynamic system control tasks]. *Zeitschrift für Experimentelle und Angewandte Psychologie*, 36, 221-238.
- Hunt, E. (1991). Some comments on the study of complexity. In R. J. Sternberg, & P. A. Frensch (Eds.), *Complex problem solving: Principles and mechanisms* (pp. 383-395). Hillsdale, NJ: Lawrence Erlbaum Associates.
- Hussy, W. (1985). Komplexes Problemlösen - Eine Sackgasse? [Complex problem solving - a dead end?]. *Zeitschrift für Experimentelle und Angewandte Psychologie*, 32, 55-77.

- Kay, D. S. (1991). Computer interaction: Debugging the problems. In R. J. Sternberg & P. A. Frensch (Eds.), *Complex problem solving: Principles and mechanisms* (pp. 317-340). Hillsdale, NJ: Lawrence Erlbaum Associates.
- Kluwe, R. H. (1993). Knowledge and performance in complex problem solving. In G. Strube & K.-F. Wender (Eds.), *The cognitive psychology of knowledge* (pp. 401-423). Amsterdam: Elsevier Science Publishers.
- Kluwe, R. H. (1995). Single case studies and models of complex problem solving. In P. A. Frensch & J. Funke (Eds.), *Complex problem solving: The European Perspective* (pp. 269-291). Hillsdale, NJ: Lawrence Erlbaum Associates.
- Kolb, S., Petzing, F., & Stumpf, S. (1992). Komplexes Problemlösen: Bestimmung der Problemlösegüte von Probanden mittels Verfahren des Operations Research ? ein interdisziplinärer Ansatz [Complex problem solving: determining the quality of human problem solving by operations research tools - an interdisciplinary approach]. *Sprache & Kognition*, 11, 115-128.
- Krems, J. F. (1995). Cognitive flexibility and complex problem solving. In P. A. Frensch & J. Funke (Eds.), *Complex problem solving: The European Perspective* (pp. 201-218). Hillsdale, NJ: Lawrence Erlbaum Associates.
- Lesgold, A., & Lajoie, S. (1991). Complex problem solving in electronics. In R. J. Sternberg & P. A. Frensch (Eds.), *Complex problem solving: Principles and mechanisms* (pp. 287-316). Hillsdale, NJ: Lawrence Erlbaum Associates.
- Mayer, R. E. (1992). *Thinking, problem solving, cognition*. Second edition. New York: W. H. Freeman and Company.
- Müller, H. (1993). *Komplexes Problemlösen: Reliabilität und Wissen* [Complex problem solving: Reliability and knowledge]. Bonn, Germany: Holos.
- Newell, A., & Simon, H. A. (1972). *Human problem solving*. Englewood Cliffs, NJ: Prentice-Hall.
- Paradies, M.W., & Unger, L. W. (2000). *TapRooT - The System for Root Cause Analysis, Problem Investigation, and Proactive Improvement*. Knoxville, TN: System Improvements.
- Putz-Osterloh, W. (1993). Strategies for knowledge acquisition and transfer of knowledge in dynamic tasks. In G. Strube & K.-F. Wender (Eds.), *The cognitive psychology of knowledge* (pp. 331-350). Amsterdam: Elsevier Science Publishers.
- Riefer, D.M., & Batchelder, W.H. (1988). Multinomial modeling and the measurement of cognitive processes. *Psychological Review*, 95, 318-339.
- Ringelband, O. J., Misiak, C., & Kluwe, R. H. (1990). Mental models and strategies in the control of a complex system. In D. Ackermann, & M. J. Tauber (Eds.), *Mental models and human-computer interaction* (Vol. 1, pp. 151-164). Amsterdam: Elsevier Science Publishers.
- Schaub, H. (1993). *Modellierung der Handlungsorganisation*. Bern, Switzerland: Hans Huber.
- Sokol, S. M., & McCloskey, M. (1991). Cognitive mechanisms in calculation. In R. J. Sternberg & P. A. Frensch (Eds.), *Complex problem solving: Principles and mechanisms* (pp. 85-116). Hillsdale, NJ: Lawrence Erlbaum Associates.
- Stanovich, K. E., & Cunningham, A. E. (1991). Reading as constrained reasoning. In R. J. Sternberg & P. A. Frensch (Eds.), *Complex problem solving: Principles and mechanisms* (pp. 3-60). Hillsdale, NJ: Lawrence Erlbaum Associates.
- Sternberg, R. J. (1995). Conceptions of expertise in complex problem solving: A comparison of alternative conceptions. In P. A. Frensch & J. Funke (Eds.), *Complex problem solving: The European Perspective* (pp. 295-321). Hillsdale, NJ: Lawrence Erlbaum Associates.
- Sternberg, R. J., & Frensch, P. A. (Eds.). (1991). *Complex problem solving: Principles and mechanisms*. Hillsdale, NJ: Lawrence Erlbaum Associates.
- Strauß, B. (1993). *Konfundierungen beim Komplexen Problemlösen. Zum Einfluß des Anteils der richtigen Lösungen (ArL) auf das Problemlöseverhalten in komplexen Situationen* [Confoundations in complex problem

solving. On the influence of the degree of correct solutions on problem solving in complex situations]. Bonn, Germany: Holos.
- Strohschneider, S. (1991). Kein System von Systemen! Kommentar zu dem Aufsatz "Systemmerkmale als Determinanten des Umgangs mit dynamischen Systemen" von Joachim Funke [No system of systems! Reply to the paper "System features as determinants of behavior in dynamic task environments" by Joachim Funke]. *Sprache & Kognition*, 10, 109-113.
- Van Lehn, K. (1989). Problem solving and cognitive skill acquisition. In M. I. Posner (Ed.), *Foundations of cognitive science* (pp. 527-579). Cambridge, MA: MIT Press.
- Voss, J. F., Wolfe, C. R., Lawrence, J. A., & Engle, R. A. (1991). From representation to decision: An analysis of problem solving in international relations. In R. J. Sternberg & P. A. Frensch (Eds.), *Complex problem solving: Principles and mechanisms* (pp. 119-158). Hillsdale, NJ: Lawrence Erlbaum Associates.
- Wagner, R. K. (1991). Managerial problem solving. In R. J. Sternberg & P. A. Frensch (Eds.), *Complex problem solving: Principles and mechanisms* (pp. 159-183). Hillsdale, NJ: Lawrence Erlbaum Associates.
- Wisconsin Educational Media Association. (1993). "Information literacy: A position paper on information problem-solving." Madison, WI: WEMA Publications. (ED 376 817). (Portions adapted from Michigan State Board of Education's Position Paper on Information Processing Skills, 1992).
- Altshuller, Genrich (1973). *Innovation Algorithm*. Worcester, MA: Technical Innovation Center. ISBN 0-9640740-2-8.
- Altshuller, Genrich (1984). *Creativity as an Exact Science*. New York, NY: Gordon & Breach. ISBN 0-677-21230-5.
- Altshuller, Genrich (1994). *And Suddenly the Inventor Appeared*. translated by Lev Shulyak. Worcester, MA: Technical Innovation Center. ISBN 0-9640740-1-X.
- D'Zurilla, T. J., & Goldfried, M. R. (1971). Problem solving and behavior modification. Journal of Abnormal Psychology, 78, 107-126.
- D'Zurilla, T. J., & Nezu, A. M. (1982). Social problem solving in adults. In P. C. Kendall (Ed.), Advances in cognitive-behavioral research and therapy (Vol. 1, pp.201–274). New York: Academic Press.
- Rath J. F.; Langenbahn D. M.; Simon D.; Sherr R. L.; Fletcher J.; Diller L. (2004). The construct of problem solving in higher level neuropsychological assessment and rehabilitation. Archives of Clinical Neuropsychology, 19, 613-635. doi:10.1016/j.acn.2003.08.006
- Rath, J. F.; Simon, D.; Langenbahn, D. M.; Sherr, R. L.; Diller, L. (2003). Group treatment of problem-solving deficits in outpatients with traumatic brain injury: A randomised outcome study. Neuropsychological Rehabilitation, 13, 461-488.

External links

- Computer Skills for Information Problem-Solving: Learning and Teaching Technology in Context [8]
- Problem solving-Elementary level [9]
- CROP (Communities Resolving Our Problems) [10]
- The Altshuller Institute for TRIZ Studies, Worcester, MA [11]

References

[1] Goldstein F. C., & Levin H. S. (1987). Disorders of reasoning and problem-solving ability. In M. Meier, A. Benton, & L. Diller (Eds.), *Neuropsychological rehabilitation*. London: Taylor & Francis Group.
[2] Duncker, K. (1935). *Zur Psychologie des produktiven Denkens* [The psychology of productive thinking]. Berlin: Julius Springer.
[3] Mayer, R. E. (1992). *Thinking, problem solving, cognition*. Second edition. New York: W. H. Freeman and Company.
[4] *Newell, A., & Simon, H. A. (1972). *Human problem solving*. Englewood Cliffs, NJ: Prentice-Hall.
[5] 2007 Draft, Washington State Revised Mathematics Standard
[6] http://www.usabilityviews.com/uv007206.html
[7] http://worldcat.org/oclc/50065032&tab=holdings

[8] http://www.ericdigests.org/1996-4/skills.htm
[9] http://moodle.ed.uiuc.edu/wiked/index.php/Problem_solving-Elementary_level
[10] http://ceap.wcu.edu/houghton/Learner/basicidea.html
[11] http://www.aitriz.org

Linear search problem

In computational complexity theory, the **Linear search problem** is an optimal search problem introduced by Richard E. Bellman.[1] (independently considered by Anatole Beck [2]).

The problem

"An immobile hider is located on the real line according to a known probability distribution. A searcher, whose maximal velocity is one, starts from the origin and wishes to discover the hider in minimal expected time. It is assumed that the searcher can change the direction of his motion without any loss of time. It is also assumed that the searcher cannot see the hider until he actually reaches the point at which the hider is located and the time elapsed until this moment is the duration of the game." It is obvious that in order to find the hider the searcher has to go a distance x_1 in one direction, return to the origin and go distance x_2 in the other direction etc., (the length of the n-th step being denoted by x_n), and to do it in an optimal way. (However, an optimal solution need not have a first step and could start with an infinite number of small 'oscillations'.) This problem is usually called the linear search problem. It has attracted much research, some of it quite recent. (Especially Beck, e.g.,[3] [4] .)

The linear search problem for a general probability distribution is unsolved yet. However, there exists a dynamic programming algorithm that produces a solution for any discrete distribution [5] and also an approximate solution, for any probability distribution, with any desired accuracy [6] .

The linear search problem was solved by Anatole Beck and Donald J. Newman (1970) as a two-person zero-sum game [7] . This solution was obtained in the framework of an online algorithm by Gal [8] . The online solution with a turn cost is given by [9] .

See also

- Search games

References

[1] R. Bellman. An optimal search problem, SIAM Rev. (1963).
[2] A. Beck. On the linear search Problem, Israel J. Mathematics (1964).
[3] A. Beck. More on the linear search problem, Israel J. Mathematics (1965).
[4] A. Beck and M. Beck. The linear search problem rides again, Israel J. Mathematics (1986).
[5] T.F. Bruce and J.B. Robertson. A survey of the linear-search problem. Math. Sci. 13 (1988).
[6] S. Alpern and S. Gal. *The Theory of Search Games and Rendezvous*. Springer (2003).
[7] A. Beck and D.J. Newman. Yet More on the linear search problem. Israel J. Math. (1970).
[8] S. Gal. SEARCH GAMES, Academic Press (1980).
[9] E. Demaine, S. Fekete and S. Gal. Online searching with turn cost. Theoretical Computer Science (2006).

Linear complementarity problem

In mathematical optimization theory, the **linear complementarity problem (LCP)** is a special case of quadratic programming which arises frequently in computational mechanics.

Formulation

Given a real matrix **M** and vector **q**, the linear complementarity problem seeks vectors **z** and **w** which satisfy the following constraints:

- $\mathbf{w} = \mathbf{M}\mathbf{z} + \mathbf{q}$
- $\mathbf{w} \geq 0, \mathbf{z} \geq 0$(that is, each component of these two vectors is non-negative)
- $w_i z_i = 0$for all i. (The complementarity condition)

A sufficient condition for existence and uniqueness of a solution to this problem is that **M** be symmetric positive-definite.

The vector $\mathbf{w}$is a slack variable, and so is generally discarded after $\mathbf{z}$is found. As such, the problem can also be formulated as:

- $\mathbf{M}\mathbf{z} + \mathbf{q} \geq 0$
- $\mathbf{z} \geq 0$
- $\mathbf{z}^{\mathrm{T}}(\mathbf{M}\mathbf{z} + \mathbf{q}) = 0$(the complementarity condition)

Relation to quadratic programming

According to the Karush–Kuhn–Tucker conditions, finding a solution to the linear complementarity problem is equivalent to minimizing the quadratic function

$$f(\mathbf{z}) = \mathbf{z}^{\mathrm{T}}(\mathbf{M}\mathbf{z} + \mathbf{q})$$

subject to the constraints

$$\mathbf{M}\mathbf{z} + \mathbf{q} \geq 0$$
$$\mathbf{z} \geq 0$$

Because these constraints ensure that *f* is always non-negative, it attains its minimum of 0 at **z** if and only if **z** solves the linear complementarity problem.

If **M** is positive definite, any algorithm for solving (convex) QPs can of course be used to solve the LCP. However, there also exist more efficient, specialized algorithms, such as Lemke's algorithm and Dantzig's algorithm.

Also, a quadratic programming problem stated as minimize $f(\mathbf{x}) = \mathbf{c}^T\mathbf{x} + \frac{1}{2}\mathbf{x}^T\mathbf{Q}\mathbf{x}$subject to $\mathbf{A}\mathbf{x} \geq 0$

is the same as solving the LCP with

- $\mathbf{q} = \begin{bmatrix} \mathbf{c} \\ -\mathbf{b} \end{bmatrix}$
- $\mathbf{M} = \begin{bmatrix} \mathbf{Q} & -\mathbf{A}^T \\ \mathbf{A} & 0 \end{bmatrix}$

In fact, most QP solvers work on the LCP formulation, including the interior point method, principal / complementarity pivoting and active set methdos.

See also

- Complementarity theory
- Physics engine Impulse/constraint type physics engines for games use this approach.

References

- Murty, K. G. (1988). *Linear complementarity, linear and nonlinear programming* [1]. Sigma Series in Applied Mathematics. **3**. Berlin: Heldermann Verlag. pp. xlviii+629 pp.. ISBN 3-88538-403-5. (Available for download at the website of Professor Katta G. Murty [2].) MR949214

Further reading

- Cottle, Richard W.; Pang, Jong-Shi; Stone, Richard E. (1992). *The linear complementarity problem*. Computer Science and Scientific Computing. Boston, MA: Academic Press, Inc.. pp. xxiv+762 pp.. ISBN 0-12-192350-9. MR1150683
- R. W. Cottle and G. B. Dantzig. Complementary pivot theory of mathematical programming. *Linear Algebra and its Applications*, 1:103-125, 1968.
- Murty, K. G. (1988). *Linear complementarity, linear and nonlinear programming* [1]. Sigma Series in Applied Mathematics. **3**. Berlin: Heldermann Verlag. pp. xlviii+629 pp.. ISBN 3-88538-403-5. (Available for download at the website of Professor Katta G. Murty [2].) MR949214

References

[1] http://ioe.engin.umich.edu/people/fac/books/murty/linear_complementarity_webbook/
[2] http://www-personal.umich.edu/~murty/

Mixed linear complementarity problem

In mathematical optimization theory, the **mixed linear complementarity problem**, often abbreviated as **MLCP** or **LMCP**, is a generalization of the linear complementarity problem to include free variables.

References

- Complementarity problems [1]
- Algorithms for complementarity problems and generalized equations [2]
- An Algorithm for the Approximate and Fast Solution of Linear Complementarity Problems [3]

References

[1] http://www.uclm.es/area/gsee/Web/Raquel/Complementarity_Problems.pdf
[2] ftp://ftp.cs.wisc.edu/math-prog/tech-reports/95-14.ps
[3] http://www.mcs.anl.gov/~leyffer/listn/slides-07/morales.pdf

Boolean satisfiability problem

Satisfiability is the problem of determining if the variables of a given Boolean formula can be assigned in such a way as to make the formula evaluate to TRUE. Equally important is to determine whether no such assignments exist, which would imply that the function expressed by the formula is identically FALSE for all possible variable assignments. In this latter case, we would say that the function is unsatisfiable; otherwise it is satisfiable. To emphasize the binary nature of this problem, it is frequently referred to as *Boolean* or *propositional satisfiability*. The shorthand "**SAT**" is also commonly used to denote it, with the implicit understanding that the function and its variables are all binary-valued.

Basic definitions, terminology and applications

In complexity theory, the **satisfiability problem** (**SAT**) is a decision problem, whose instance is a Boolean expression written using only AND, OR, NOT, variables, and parentheses. The question is: given the expression, is there some assignment of *TRUE* and *FALSE* values to the variables that will make the entire expression true? A formula of propositional logic is said to be *satisfiable* if logical values can be assigned to its variables in a way that makes the formula true. The boolean satisfiability problem is NP-complete. The propositional satisfiability problem (PSAT), which decides whether a given propositional formula is satisfiable, is of central importance in various areas of computer science, including theoretical computer science, algorithmics, artificial intelligence, hardware design, electronic design automation, and verification.

A **literal** is either a variable or the negation of a variable (the negation of an expression can be reduced to negated variables by De Morgan's laws). For example, $x_1$is a **positive literal** and $not(x_2)$is a **negative literal**.

A **clause** is a disjunction of literals. For example, $x_1 \vee not(x_2)$is a clause.

There are several special cases of the Boolean satisfiability problem in which the formulae are required to be conjunctions of clauses (i.e. formulae in conjunctive normal form). Determining the satisfiability of a formula in conjunctive normal form where each clause is limited to at most three literals is NP-complete; this problem is called "3SAT", "3CNFSAT", or "3-satisfiability". Determining the satisfiability of a formula in which each clause is limited to at most two literals is NL-complete; this problem is called "2SAT". Determining the satisfiability of a formula in which each clause is a Horn clause (i.e. it contains at most one positive literal) is P-complete; this problem is called Horn-satisfiability.

The Cook–Levin theorem proves that the Boolean satisfiability problem is NP-complete, and in fact, this was the first decision problem proved to be NP-complete. However, beyond this theoretical significance, efficient and scalable algorithms for SAT that were developed over the last decade have contributed to dramatic advances in our ability to automatically solve problem instances involving tens of thousands of variables and millions of constraints. Examples of such problems in electronic design automation (EDA) include formal equivalence checking, model checking, formal verification of pipelined microprocessors, automatic test pattern generation, routing of FPGAs, and so on. A SAT-solving engine is now considered to be an essential component in the EDA toolbox.

Complexity and restricted versions

NP-completeness

SAT was the first known NP-complete problem, as proved by Stephen Cook in 1971 (see Cook's theorem for the proof). Until that time, the concept of an NP-complete problem did not even exist. The problem remains **NP**-complete even if all expressions are written in *conjunctive normal form* with 3 variables per clause (3-CNF), yielding the **3SAT** problem. This means the expression has the form:

$(x_{11}$ OR x_{12} OR $x_{13})$ AND
$(x_{21}$ OR x_{22} OR $x_{23})$ AND
$(x_{31}$ OR x_{32} OR $x_{33})$ AND ...

where each x is a variable or a negation of a variable, and each variable can appear multiple times in the expression.

A useful property of Cook's reduction is that it preserves the number of accepting answers. For example, if a graph has 17 valid 3-colorings, the SAT formula produced by the reduction will have 17 satisfying assignments.

NP-completeness only refers to the run-time of the worst case instances. Many of the instances that occur in practical applications can be solved much more quickly. See runtime behavior below.

SAT is easier if the formulas are restricted to those in disjunctive normal form, that is, they are disjunction (OR) of terms, where each term is a conjunction (AND) of literals (possibly negated variables). Such a formula is indeed satisfiable if and only if at least one of its terms is satisfiable, and a term is satisfiable if and only if it does not contain both x and NOT x for some variable x. This can be checked in polynomial time.

2-satisfiability

SAT is also easier if the number of literals in a clause is limited to 2, in which case the problem is called 2SAT. This problem can also be solved in polynomial time, and in fact is complete for the class NL. Similarly, if we limit the number of literals per clause to 2 and change the AND operations to XOR operations, the result is *exclusive-or 2-satisfiability*, a problem complete for SL = L.

One of the most important restrictions of SAT is HORNSAT, where the formula is a conjunction of Horn clauses. This problem is solved by the polynomial-time Horn-satisfiability algorithm, and is in fact P-complete. It can be seen as P's version of the Boolean satisfiability problem.

Provided that the complexity classes P and NP are not equal, none of these restrictions are NP-complete, unlike SAT. The assumption that P and NP are not equal is currently not proven.

3-satisfiability

3-satisfiability is a special case of *k*-satisfiability (*k*-SAT) or simply satisfiability (SAT), when each clause contains exactly *k* = 3 literals. It was one of Karp's 21 NP-complete problems.

Here is an example, where ¬ indicates negation:

E = (x_1 or ¬x_2 or ¬x_3) and (x_1 or x_2 or x_4)

E has two clauses (denoted by parentheses), four variables (x_1, x_2, x_3, x_4), and *k*=3 (three literals per clause).

To solve this instance of the decision problem we must determine whether there is a truth value (TRUE or FALSE) we can assign to each of the literals (x_1 through x_4) such that the entire expression is TRUE. In this instance, there is such an assignment (x_1 = TRUE, x_2 = TRUE, x_3=TRUE, x_4=TRUE), so the answer to this instance is YES. This is one of many possible assignments, with for instance, any set of assignments including x_1 = TRUE being sufficient. If there were no such assignment(s), the answer would be NO.

Since k-SAT (the general case) reduces to 3-SAT, and 3-SAT can be proven to be NP-complete, it can be used to prove that other problems are also NP-complete. This is done by showing how a solution to another problem could be used to solve 3-SAT. An example of a problem where this method has been used is "Clique". It's often easier to use reductions from 3-SAT than SAT to problems that researchers are attempting to prove NP-complete.

3-SAT can be further restricted to One-in-three 3SAT, where we ask if *exactly* one of the variables in each clause is true, rather than *at least* one. One-in-three 3SAT remains NP-complete.

Horn-satisfiability

A clause is Horn if it contains at most one positive literal. Such clauses are of interest because they are able to express implication of one variable from a set of other variables. Indeed, one such clause $\neg x_1 \vee \cdots \vee \neg x_n \vee y$ can be rewritten as $x_1 \wedge \cdots \wedge x_n \rightarrow y$, that is, if $x_1, \ldots, x_n$ are all true, then y needs to be true as well.

The problem of deciding whether a set of Horn clauses is satisfiable is in P. This problem can indeed be solved by a single step of the Unit propagation, which produces the single minimal (w.r.t. the set of literal assigned to true) model of the set of Horn clauses.

A generalization of the class of Horn formulae is that of renamable-Horn formulae, which is the set of formulae that can be placed in Horn form by replacing some variables with their respective negation. Checking the existence of such a replacement can be done in linear time; therefore, the satisfiability of such formulae is in P as it can be solved by first performing this replacement and then checking the satisfiability of the resulting Horn formula.

Schaefer's dichotomy theorem

The restrictions above (CNF, 2CNF, 3CNF, Horn) bound the considered formulae to be conjunction of subformulae; each restriction states a specific form for all subformulae: for example, only binary clauses can be subformulae in 2CNF.

Schaefer's dichotomy theorem states that, for any restriction to Boolean operators that can be used to form these subformulae, the corresponding satisfiability problem is in P or NP-complete. The membership in P of the satisfiability of 2CNF and Horn formulae are special cases of this theorem.

Runtime behavior

As mentioned briefly above, though the problem is NP-complete, many practical instances can be solved much more quickly. Many practical problems are actually "easy", so the SAT solver can easily find a solution, or prove that none exists, relatively quickly, even though the instance has thousands of variables and tens of thousands of constraints. Other much smaller problems exhibit run-times that are exponential in the problem size, and rapidly become impractical. Unfortunately, there is no reliable way to tell the difficulty of the problem without trying it. Therefore,

almost all SAT solvers include time-outs, so they will terminate even if they cannot find a solution. Finally, different SAT solvers will find different instances easy or hard, and some excel at proving unsatisfiability, and others at finding solutions. All of these behaviors can be seen in the SAT solving contests.[1]

Extensions of SAT

An extension that has gained significant popularity since 2003 is Satisfiability modulo theories that can enrich CNF formulas with linear constraints, arrays, all-different constraints, uninterpreted functions, etc. Such extensions typically remain NP-complete, but very efficient solvers are now available that can handle many such kinds of constraints.

The satisfiability problem becomes more difficult (PSPACE-complete) if we extend our logic to include second-order Booleans, allowing *quantifiers* such as "for all" and "there exists" that bind the Boolean variables. An example of such an expression would be:

$$\forall x \left(\exists y \left(\exists z \left((x \vee y \vee z) \wedge (\neg x \vee \neg y \vee \neg z)\right)\right)\right).$$

SAT itself uses only $\exists$ quantifiers. If we allow only $\forall$ quantifiers, it becomes the Co-NP-complete tautology problem. If we allow both, the problem is called the quantified Boolean formula problem (QBF), which can be shown to be PSPACE-complete. It is widely believed that PSPACE-complete problems are strictly harder than any problem in NP, although this has not yet been proved.

A number of variants deal with the number of variable assignments making the formula true. Ordinary SAT asks if there is at least one such assignment. MAJSAT, which asks if the majority of all assignments make the formula true, is complete for PP, a probabilistic class. The problem of how many variable assignments satisfy a formula, not a decision problem, is in #P. UNIQUE-SAT or USAT is the problem of determining whether a formula known to have either zero or one satisfying assignments has zero or has one. Although this problem seems easier, it has been shown that if there is a practical (randomized polynomial-time) algorithm to solve this problem, then all problems in NP can be solved just as easily.

The maximum satisfiability problem, an FNP generalization of SAT, asks for the maximum number of clauses which can be satisfied by any assignment. It has efficient approximation algorithms, but is NP-hard to solve exactly. Worse still, it is APX-complete, meaning there is no polynomial-time approximation scheme (PTAS) for this problem unless P=NP.

Algorithms for solving SAT

There are two classes of high-performance algorithms for solving instances of SAT in practice: modern variants of the DPLL algorithm, such as Chaff, GRASP or march [2], and stochastic local search algorithms, such as WalkSAT.

A DPLL SAT solver employs a systematic backtracking search procedure to explore the (exponentially-sized) space of variable assignments looking for satisfying assignments. The basic search procedure was proposed in two seminal papers in the early 60s (see references below) and is now commonly referred to as the Davis-Putnam-Logemann-Loveland algorithm (“DPLL” or “DLL”). Theoretically, exponential lower bounds have been proved for the DPLL family of algorithms.

Modern SAT solvers (developed in the last ten years) come in two flavors: "conflict-driven" and "look-ahead". Conflict-driven solvers augment the basic DPLL search algorithm with efficient conflict analysis, clause learning, non-chronological backtracking (aka backjumping), as well as "two-watched-literals" unit propagation, adaptive branching, and random restarts. These "extras" to the basic systematic search have been empirically shown to be essential for handling the large SAT instances that arise in Electronic Design Automation (EDA). Look-ahead solvers have especially strengthened reductions (going beyond unit-clause propagation) and the heuristics, and they are generally stronger than conflict-driven solvers on hard instances (while conflict-driven solvers can be much better on large instances which have inside actually an easy instance).

Modern SAT solvers are also having significant impact on the fields of software verification, constraint solving in artificial intelligence, and operations research, among others. Powerful solvers are readily available as free and open source software. In particular, the conflict-driven MiniSAT [3], which was relatively successful at the 2005 SAT competition [4], only has about 600 lines of code. An example for look-ahead solvers is march_dl [2], which won a prize at the 2007 SAT competition [4].

Genetic algorithms and other general-purpose stochastic local search methods are also being used to solve SAT problems, especially when there is no or limited knowledge of the specific structure of the problem instances to be solved. Certain types of large random satisfiable instances of SAT can be solved by survey propagation (SP). Particularly in hardware design and verification applications, satisfiability and other logical properties of a given propositional formula are sometimes decided based on a representation of the formula as a binary decision diagram (BDD).

Propositional satisfiability has various generalisations, including satisfiability for quantified Boolean formula problem, for first- and second-order logic, constraint satisfaction problems, 0-1 integer programming, and maximum satisfiability problem.

Many other decision problems, such as graph coloring problems, planning problems, and scheduling problems, can be easily encoded into SAT.

See also

- Unsatisfiable core
- Satisfiability Modulo Theories

References

References are listed in order by date of publishing:

- M. Davis and H. Putnam, A Computing Procedure for Quantification Theory (doi:10.1145/321033.321034), Journal of the Association for Computing Machinery, vol. 7, no. 3, pp. 201–215, 1960.
- M. Davis, G. Logemann, and D. Loveland, A Machine Program for Theorem-Proving (doi:10.1145/368273.368557), Communications of the ACM, vol. 5, no. 7, pp. 394–397, 1962.
- S. A. Cook, The Complexity of Theorem Proving Procedures (doi:10.1145/800157.805047), in Proc. 3rd Ann. ACM Symp. on Theory of Computing, pp. 151–158, Association for Computing Machinery, 1971.
- Michael R. Garey and David S. Johnson (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman. ISBN 0-7167-1045-5. A9.1: LO1 – LO7, pp. 259 – 260.
- J. P. Marques-Silva and K. A. Sakallah, GRASP: A Search Algorithm for Propositional Satisfiability (doi:10.1109/12.769433), IEEE Transactions on Computers, vol. 48, no. 5, pp. 506–521, 1999.
- J.-P. Marques-Silva and T. Glass, Combinational Equivalence Checking Using Satisfiability and Recursive Learning (doi:10.1109/DATE.1999.761110), in Proc. Design, Automation and Test in Europe Conference, pp. 145–149, 1999.
- R. E. Bryant, S. M. German, and M. N. Velev, Microprocessor Verification Using Efficient Decision Procedures for a Logic of Equality with Uninterpreted Functions [5], in Analytic Tableaux and Related Methods, pp. 1–13, 1999.
- M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, Chaff: engineering an efficient SAT solver (doi:10.1145/378239.379017), in Proc. 38th ACM/IEEE Design Automation Conference, pp. 530–535, Las Vegas, Nevada, 2001.
- E. Clarke [6], A. Biere [7], R. Raimi, and Y. Zhu, Bounded Model Checking Using Satisfiability Solving (doi:10.1023/A:1011276507260), Formal Methods in System Design, vol. 19, no. 1, 2001.

- M. Perkowski and A. Mishchenko, "Logic Synthesis for Regular Layout using Satisfiability," in Proc. Intl Workshop on Boolean Problems, 2002.
- G.-J. Nam, K. A. Sakallah, and R. Rutenbar, A New FPGA Detailed Routing Approach via Search-Based Boolean Satisfiability (doi:10.1109/TCAD.2002.1004311), IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 21, no. 6, pp. 674–684, 2002.
- N. Een and N. Sörensson, An Extensible SAT-solver (doi:10.1007/b95238), in Satisfiability Workshop, 2003.
- D. Babić, J. Bingham, and A. J. Hu, B-Cubing: New Possibilities for Efficient SAT-Solving (doi:10.1109/TC.2006.175), IEEE Transactions on Computers 55(11):1315–1324, 2006.
- C. Rodríguez, M. Villagra and B. Barán, Asynchronous team algorithms for Boolean Satisfiability (doi:10.1109/BIMNICS.2007.4610083), Bionetics2007, pp. 66–69, 2007.

[1] "The international SAT Competitions web page" (http://www.satcompetition.org/). . Retrieved 2007-11-15.
[2] http://www.st.ewi.tudelft.nl/sat/march_dl.php
[3] http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat/
[4] http://www.satcompetition.org/
[5] http://portal.acm.org/citation.cfm?id=709275
[6] http://www.cs.cmu.edu/~emc/
[7] http://fmv.jku.at/biere/

External links

More information on SAT:

- SAT and MAX-SAT for the Lay-researcher (http://users.ecs.soton.ac.uk/mqq06r/sat/)

SAT Applications:

- WinSAT v2.04 (http://users.ecs.soton.ac.uk/mqq06r/winsat/): A Windows-based SAT application made particularly for researchers.

SAT Solvers:

- Chaff (http://www.princeton.edu/~chaff/)
- HyperSAT (http://www.domagoj-babic.com/index.php/ResearchProjects/HyperSAT)
- Spear (http://www.domagoj-babic.com/index.php/ResearchProjects/Spear)
- The MiniSAT Solver (http://minisat.se/)
- UBCSAT (http://www.satlib.org/ubcsat/)
- Sat4j (http://www.sat4j.org/)
- RSat (http://reasoning.cs.ucla.edu/rsat/home.html)
- Fast SAT Solver (http://dudka.cz/fss) - simple but fast implementation of SAT solver based on genetic algorithms
- CVC3 (http://www.cs.nyu.edu/acsys/cvc3/)

Conferences/Publications:

- SAT 2009: Twelfth International Conference on Theory and Applications of Satisfiability Testing (http://www.cs.swansea.ac.uk/~csoliver/SAT2009/)
- SAT 2008: Eleventh International Conference on Theory and Applications of Satisfiability Testing (http://wwwcs.uni-paderborn.de/cs/ag-klbue/en/workshops/sat-08/sat08-main.php)
- SAT 2007: Tenth International Conference on Theory and Applications of Satisfiability Testing (http://sat07.ecs.soton.ac.uk/)
- Journal on Satisfiability, Boolean Modeling and Computation (http://jsat.ewi.tudelft.nl)
- Survey Propagation (http://www.ictp.trieste.it/~zecchina/SP/)

Benchmarks:

- Forced Satisfiable SAT Benchmarks (http://www.nlsde.buaa.edu.cn/~kexu/benchmarks/benchmarks.htm)

- IBM Formal Verification SAT Benchmarks (http://www.haifa.il.ibm.com/projects/verification/RB_Homepage/bmcbenchmarks.html)
- SATLIB (http://www.satlib.org)
- Software Verification Benchmarks (http://www.cs.ubc.ca/~babic/index_benchmarks.htm)
- Fadi Aloul SAT Benchmarks (http://www.aloul.net/benchmarks.html)

SAT solving in general:

- http://www.satlive.org
- http://www.satisfiability.org

Evaluation of SAT solvers:

- Yearly evaluation of SAT solvers (http://www.maxsat.udl.cat/)
- SAT solvers evaluation results for 2008 (http://www.maxsat.udl.cat/08/ms08.pdf)

This article includes material from a column in the ACM SIGDA (http://www.sigda.org) *e-newsletter* (http://www.sigda.org/newsletter/index.html) *by Prof. Karem Sakallah* (http://www.eecs.umich.edu/~karem)
Original text is available here (http://www.sigda.org/newsletter/2006/eNews_061201.html)

Mathematical problem

A **mathematical problem** is a problem that is amenable to being analyzed, and possibly solved, with the methods of mathematics. This can be a real-world problem, such as computing the orbits of the planets in the solar system, or a problem of a more abstract nature, such as Hilbert's problems. It can also be a problem referring to the nature of mathematics itself, such as Russell's Paradox.

Real-world problems

Informal "real-world" mathematical problems are questions related to a concrete setting, such as "Adam has five apples and gives John three. How many has he left?". Such questions are usually more difficult to solve than regular mathematical exercises like "5 – 3", even if one knows the mathematics required to solve the problem. Known as word problems, they are used in mathematics education to teach students to connect real-world situations to the abstract language of mathematics.

In general, to use mathematics for solving a real-world problem, the first step is to construct a mathematical model of the problem. This involves abstraction from the details of the problem, and the modeller has to be careful not to lose essential aspects in translating the original problem into a mathematical one. After the problem has been solved in the world of mathematics, the solution must be translated back into the context of the original problem.

Abstract problems

Abstract mathematical problems arise in all fields of mathematics. While mathematicians usually study them for their own sake, by doing so results may be obtained that find application outside the realm of mathematics. Theoretical physics has historically been, and remains, a rich source of inspiration.

Some abstract problems have been rigorously proved to be unsolvable, such as squaring the circle and trisecting the angle using only the compass and straightedge constructions of classical geometry, and solving the general quintic equation algebraically. Also provably unsolvable are so-called undecidable problems, such as the halting problem for Turing machines.

Many abstract problems can be solved routinely, others have been solved with great effort, for some significant inroads have been made without having led yet to a full solution, and yet others have withstood all attempts, such as

Goldbach's conjecture and the Collatz conjecture. Some well-known difficult abstract problems that have been solved relatively recently are the four-colour theorem, Fermat's Last Theorem, and the Poincaré conjecture.

See also

- Mathematical problems
- Problem solving
- Mathematics game

External links

- Historical Math Problems [1] at Convergence [2]

References

[1] http://mathdl.maa.org/convergence/1/?pa=content&sa=browseNode&categoryId=9
[2] http://mathdl.maa.org/convergence/1/

Travelling salesman problem

The **Travelling Salesman Problem** (**TSP**) is a problem in combinatorial optimization studied in operations research and theoretical computer science. Given a list of cities and their pairwise distances, the task is to find a shortest possible tour that visits each city exactly once.

An optimal TSP tour through Germany's 15 largest cities. It is the shortest among 43 589 145 600[1] possible tours visiting each city exactly once.

The problem was first formulated as a mathematical problem in 1930 and is one of the most intensively studied problems in optimization. It is used as a benchmark for many optimization methods. Even though the problem is computationally difficult, a large number of heuristics and exact methods are known, so that some instances with tens of thousands of cities can be solved.

The TSP has several applications even in its purest formulation, such as planning, logistics, and the manufacture of microchips. Slightly modified, it appears as a sub-problem in many areas, such as DNA sequencing. In these applications, the concept *city* represents, for example, customers, soldering points, or DNA fragments, and the concept *distance* represents travelling times or cost, or a similarity measure between DNA fragments. In many applications, additional constraints such as limited resources or time windows make the problem considerably harder.

In the theory of computational complexity, the decision version of TSP belongs to the class of NP-complete problems. Thus, it is assumed that there is no efficient algorithm for solving TSPs. In other words, it is likely that the worst case running time for any algorithm for TSP increases exponentially with the number of cities, so even some instances with only hundreds of cities will take many CPU years to solve exactly.

History

The origins of the travelling salesman problem are unclear. A handbook for travelling salesmen from 1832 mentions the problem and includes example tours through Germany and Switzerland, but contains no mathematical treatment.[2]


William Rowan Hamilton

Mathematical problems related to the travelling salesman problem were treated in the 1800s by the Irish mathematician W. R. Hamilton and by the British mathematician Thomas Kirkman. Hamilton's Icosian Game was a recreational puzzle based on finding a Hamiltonian cycle.[3] The general form of the TSP appears to have been first studied by mathematicians during the 1930s in Vienna and at Harvard, notably by Karl Menger, who defines the problem, considers the obvious brute-force algorithm, and observes the non-optimality of the nearest neighbour heuristic:

> We denote by *messenger problem* (since in practice this question should be solved by each postman, anyway also by many travelers) the task to find, for finitely many points whose pairwise distances are known, the shortest route connecting the points. Of course, this problem is solvable by finitely many trials. Rules which would push the number of trials below the number of permutations of the given points, are not known. The rule that one first should go from the starting point to the closest point, then to the point closest to this, etc., in general does not yield the shortest route. [4]

Hassler Whitney at Princeton University introduced the name *travelling salesman problem* soon after.[5]

In the 1950s and 1960s, the problem became increasingly popular in scientific circles in Europe and the USA. Notable contributions were made by George Dantzig, Delbert Ray Fulkerson and Selmer M. Johnson at the RAND Corporation in Santa Monica, who expressed the problem as an integer linear program and developed the cutting plane method for its solution. With these new methods they solved an instance with 49 cities to optimality by constructing a tour and proving that no other tour could be shorter. In the following decades, the problem was studied by many researchers from mathematics, computer science, chemistry, physics, and other sciences.

Richard M. Karp showed in 1972 that the Hamiltonian cycle problem was NP-complete, which implies the NP-hardness of TSP. This supplied a mathematical explanation for the apparent computational difficulty of finding optimal tours.

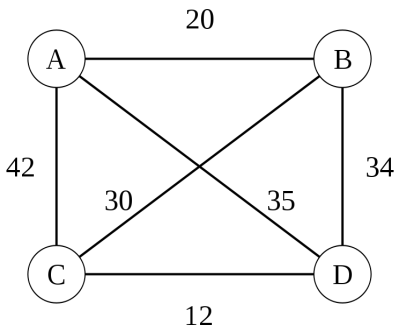
Great progress was made in the late 1970s and 1980, when Grötschel, Padberg, Rinaldi and other managed to exactly solve instances with up to 2392 cities, using cutting planes and branch-and-bound.

In the 1990s, Applegate, Bixby, Chvátal, and Cook developed the program *Concorde* that has been used in many recent record solutions. Gerhard Reinelt published the TSPLIB in 1991, a collection of benchmark instances of varying difficulty, which has been used by many research groups for comparing results. In 2005, Cook and others computed an optimal tour through a 33,810-city instance given by a microchip layout problem, currently the largest solved TSPLIB instance. For many other instances with millions of cities, solutions can be found that are provably within 1% of optimal tour.

Description

As a graph problem

TSP can be modeled as a graph, such that cities are the graph's vertices, paths are the graph's edges, and a path's distance is the edge's length. A TSP tour becomes a Hamiltonian cycle, and the optimal TSP tour is the shortest Hamiltonian cycle. Often, the model is a complete graph (*i.e.*, an edge connects each pair of vertices). If no path exists between two cities, adding an arbitrarily long edge will complete the graph without affecting the optimal tour.



Symmetric TSP with four cities

Asymmetric and symmetric

In the *symmetric TSP*, the distance between two cities is the same in each opposite direction, forming an undirected graph. This symmetry halves the number of possible solutions. In the *asymmetric TSP*, paths may not exist in both directions or the distances might be different, forming a directed graph. Traffic collisions, one-way streets, and airfares for cities with different departure and arrival fees are examples of how this symmetry could break down.

With metric distances

In the *metric TSP*, also known as *delta-TSP* or Δ-TSP, the intercity distances satisfy the triangle inequality. This can be understood as “no shortcuts”, in the sense that the direct connection from A to B is never longer than the detour via C:

$$c_{ij} \leq c_{ik} + c_{kj}$$

These edge lengths define a metric on the set of vertices. When the cities are viewed as points in the plane, many natural distance functions are metrics.

- In the Euclidian TSP the distance between two cities is the Euclidean distance between the corresponding points.
- In the Rectilinear TSP the distance between two cities is the sum of the differences of their *x*- and *y*-coordinates. This metric is often called the Manhattan distance or *city-block metric*.
- In the maximum metric, the distance between two points is the maximum of the differences of their *x*- and *y*-coordinates.

The last two metrics appear for example in routing a machine that drills a given set of holes in a printed circuit board. The Manhattan metric corresponds to a machine that adjusts first one co-ordinate, and then the other, so the time to move to a new point is the sum of both movements. The maximum metric corresponds to a machine that adjusts both co-ordinates simultaneously, so the time to move to a new point is the slower of the two movements.

Despite the simplification over the general case, the metric TSP problem with an arbitrary metric is still NP-complete.

Non-metric distances

Distance measures that do not satisfy the triangle inequality arise in many routing problems. For example, one mode of transportation, such as travel by airplane, may be faster, even though it covers a longer distance.

In its definition, the TSP does not allow cities to be visited twice, but many applications do not need this constraint. In such cases, a symmetric, non-metric instance can be reduced to a metric one. This replaces the original graph with a complete graph in which the inter-city distance c_{ij} is replaced by the shortest path between i and j in the original graph.

Related problems

- An equivalent formulation in terms of graph theory is: Given a complete weighted graph (where the vertices would represent the cities, the edges would represent the roads, and the weights would be the cost or distance of that road), find a Hamiltonian cycle with the least weight.
- The requirement of returning to the starting city does not change the computational complexity of the problem, see Hamiltonian path problem.
- Another related problem is the bottleneck travelling salesman problem (bottleneck TSP): Find a Hamiltonian cycle in a weighted graph with the minimal weight of the weightiest edge. The problem is of considerable practical importance, apart from evident transportation and logistics areas. A classic example is in printed circuit manufacturing: scheduling of a route of the drill machine to drill holes in a PCB. In robotic machining or drilling applications, the "cities" are parts to machine or holes (of different sizes) to drill, and the "cost of travel" includes time for retooling the robot (single machine job sequencing problem).
- The generalized travelling salesman problem deals with "states" that have (one or more) "cities" and the salesman has to visit exactly one "city" from each "state". Also known as the "travelling politician problem". One application is encountered in ordering a solution to the cutting stock problem in order to minimise knife changes. Another is concerned with drilling in semiconductor manufacturing, see e.g. U.S. Patent 7054798 [6]. Surprisingly, Behzad and Modarres[7] demonstrated that the generalised travelling salesman problem can be transformed into a standard travelling salesman problem with the same number of cities, but a modified distance matrix.
- The sequential ordering problem deals with the problem of visiting a set of cities where precedence relations between the cities exist.
- The travelling purchaser problem deals with a purchaser who is charged with purchasing a set of products. He can purchase these products in several cities, but at different prices and not all cities offer the same products. The objective is to to find a route between a subset of the cities, which minimizes total cost (travel cost + purchasing cost) and which enables the purchase of all required products.

Computing a solution

The traditional lines of attack for the NP-hard problems are the following:

- Devising algorithms for finding exact solutions (they will work reasonably fast only for relatively small problem sizes).
- Devising "suboptimal" or heuristic algorithms, i.e., algorithms that deliver either seemingly or probably good solutions, but which could not be proved to be optimal.
- Finding special cases for the problem ("subproblems") for which either better or exact heuristics are possible.

Computational complexity

The problem has been shown to be NP-hard (more precisely, it is complete for the complexity class FP^{NP}; see function problem), and the decision problem version ("given the costs and a number *x*, decide whether there is a round-trip route cheaper than *x*") is NP-complete. The bottleneck travelling salesman problem is also NP-hard. The problem remains NP-hard even for the case when the cities are in the plane with Euclidean distances, as well as in a number of other restrictive cases. Removing the condition of visiting each city "only once" does not remove the NP-hardness, since it is easily seen that in the planar case there is an optimal tour that visits each city only once (otherwise, by the triangle inequality, a shortcut that skips a repeated visit would not increase the tour length).

Complexity of approximation

In the general case, finding a shortest travelling salesman tour is NPO-complete.[8] If the distance measure is a metric and symmetric, the problem becomes APX-complete[9] and Christofides's algorithm approximates it within 3/2.[10] If the distances are restricted to 1 and 2 (but still are a metric) the approximation ratio becomes 7/6. In the asymmetric, metric case, only logarithmic performance guarantees are known, the best current algorithm achieves performance ratio 0.814 log *n*;[11] it is an open question if a constant factor approximation exists.

The corresponding maximization problem of finding the *longest* travelling salesman tour is approximable within 63/38.[12] If the distance function is symmetric, the longest tour can be approximated within 4/3 by a deterministic algorithm [13] and within $(33+\epsilon)/25$ by a randomised algorithm.[14]

Exact algorithms

The most direct solution would be to try all permutations (ordered combinations) and see which one is cheapest (using brute force search). The running time for this approach lies within a polynomial factor of $O(n!)$, the factorial of the number of cities, so this solution becomes impractical even for only 20 cities. One of the earliest applications of dynamic programming is an algorithm that solves the problem in time $O(n^2 2^n)$.[15]
The dynamic programming solution requires exponential space. Using inclusion–exclusion, the problem can be solved in time within a polynomial factor of 2^n and polynomial space.[16]

Improving these time bounds seems to be difficult. For example, it is an open problem if there exists an exact algorithm for TSP that runs in time $O(1.9999^n)$[17]

Other approaches include:

- Various branch-and-bound algorithms, which can be used to process TSPs containing 40-60 cities.
- Progressive improvement algorithms which use techniques reminiscent of linear programming. Works well for up to 200 cities.
- Implementations of branch-and-bound and problem-specific cut generation; this is the method of choice for solving large instances. This approach holds the current record, solving an instance with 85,900 cities, see Applegate (2006)

An exact solution for 15,112 German towns from TSPLIB was found in 2001 using the cutting-plane method proposed by George Dantzig, Ray Fulkerson, and Selmer Johnson in 1954, based on linear programming. The computations were performed on a network of 110 processors located at Rice University and Princeton University (see the Princeton external link). The total computation time was equivalent to 22.6 years on a single 500 MHz Alpha processor. In May 2004, the travelling salesman problem of visiting all 24,978 towns in Sweden was solved: a tour of length approximately 72,500 kilometers was found and it was proven that no shorter tour exists.[18]

In March 2005, the travelling salesman problem of visiting all 33,810 points in a circuit board was solved using *Concorde TSP Solver*: a tour of length 66,048,945 units was found and it was proven that no shorter tour exists. The computation took approximately 15.7 CPU years (Cook et al. 2006). In April 2006 an instance with 85,900 points was solved using *Concorde TSP Solver*, taking over 136 CPU years, see Applegate (2006).

Heuristic and approximation algorithms

Various heuristics and approximation algorithms, which quickly yield good solutions have been devised. Modern methods can find solutions for extremely large problems (millions of cities) within a reasonable time which are with a high probability just 2-3% away from the optimal solution.

Several categories of heuristics are recognized.

Constructive heuristics

The nearest neighbour (NN) algorithm (or so-called greedy algorithm) lets the salesman choose the nearest unvisited city as his next move. This algorithm quickly yields an effectively short route. For N cities randomly distributed on a plane, the algorithm on average yields length = 1.25 * exact_shortest_length. However, there exist many specially arranged city distributions which make the NN algorithm give the worst route (Gutin, Yeo, and Zverovich, 2002). This is true for both asymmetric and symmetric TSPs (Gutin and Yeo, 2007). Rosenkrantz et al. [1977] showed that the NN algorithm has the approximation factor $\Theta(\log |V|)$ for instances satisfying the triangle inequality.

The bitonic tour of a set of points is the minimum-perimeter monotone polygon that has the points as its vertices; it can be computed efficiently by dynamic programming.

Another constructive heuristic, Match Twice and Stitch (MTS) (Kahng, Reda 2004 [19]), MTS performs two sequential matchings, where the second matching is executed after deleting all the edges of the first matching, to yield a set of cycles. The cycles are then stitched to produce the final tour.

Iterative improvement

- **Pairwise exchange**, or **Lin–Kernighan** heuristics.

  The pairwise exchange or '2-opt' technique involves iteratively removing two edges and replacing these with two different edges that reconnect the fragments created by edge removal into a new and shorter tour. This is a special case of the k-opt method. Note that the label 'Lin–Kernighan' is an often heard misnomer for 2-opt. Lin–Kernighan is actually a more general method.

- ***k*-opt heuristic**

  Take a given tour and delete k mutually disjoint edges. Reassemble the remaining fragments into a tour, leaving no disjoint subtours (that is, don't connect a fragment's endpoints together). This in effect simplifies the TSP under consideration into a much simpler problem. Each fragment endpoint can be connected to $2k - 2$ other possibilities: of $2k$ total fragment endpoints available, the two endpoints of the fragment under consideration are disallowed. Such a constrained $2k$-city TSP can then be solved with brute force methods to find the least-cost recombination of the original fragments. The k-opt technique is a special case of the V-opt or variable-opt technique. The most popular of the k-opt methods are 3-opt, and these were introduced by Shen Lin of Bell Labs in 1965. There is a special case of 3-opt where the edges are not disjoint (two of the edges are adjacent to one another). In practice, it is often possible to achieve substantial improvement over 2-opt without the combinatorial cost of the general 3-opt by restricting the 3-changes to this special subset where two of the removed edges are adjacent. This so-called two-and-a-half-opt typically falls roughly midway between 2-opt and 3-opt, both in terms of the quality of tours achieved and the time required to achieve those tours.

- ***V*-opt heuristic**

  The variable-opt method is related to, and a generalization of the k-opt method. Whereas the k-opt methods remove a fixed number (k) of edges from the original tour, the variable-opt methods do not fix the size of the edge set to remove. Instead they grow the set as the search process continues. The best known method in this family is the Lin–Kernighan method (mentioned above as a misnomer for 2-opt). Shen Lin and Brian Kernighan first published their method in 1972, and it was the most reliable heuristic for solving travelling salesman problems for nearly two decades. More advanced variable-opt methods were developed at Bell Labs

in the late 1980s by David Johnson and his research team. These methods (sometimes called Lin–Kernighan–Johnson) build on the Lin–Kernighan method, adding ideas from tabu search and evolutionary computing. The basic Lin–Kernighan technique gives results that are guaranteed to be at least 3-opt. The Lin–Kernighan–Johnson methods compute a Lin–Kernighan tour, and then perturb the tour by what has been described as a mutation that removes at least four edges and reconnecting the tour in a different way, then v-opting the new tour. The mutation is often enough to move the tour from the local minimum identified by Lin–Kernighan. V-opt methods are widely considered the most powerful heuristics for the problem, and are able to address special cases, such as the Hamilton Cycle Problem and other non-metric TSPs that other heuristics fail on. For many years Lin–Kernighan–Johnson had identified optimal solutions for all TSPs where an optimal solution was known and had identified the best known solutions for all other TSPs on which the method had been tried.

Randomised improvement

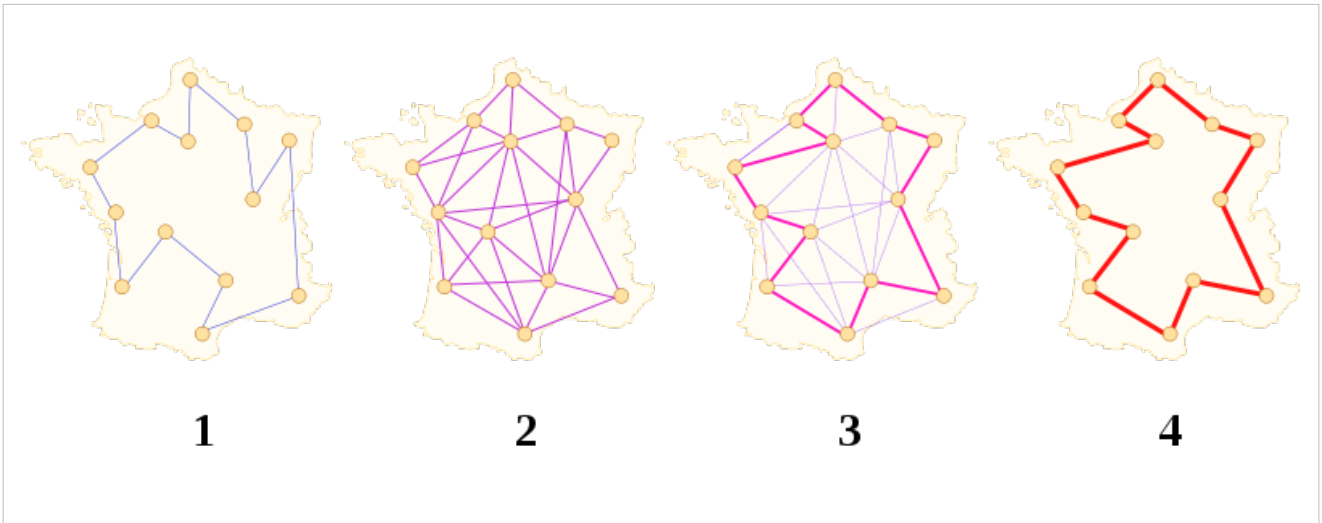
- Optimized Markov chain algorithms which use local searching heuristic sub-algorithms can find a route extremely close to the optimal route for 700 to 800 cities.
- Random path change algorithms are currently the state-of-the-art search algorithms and work up to 100,000 cities. The concept is quite simple: Choose a random path, choose four nearby points, swap their ways to create a new random path, while in parallel decreasing the upper bound of the path length. If repeated until a certain number of trials of random path changes fail due to the upper bound, one has found a local minimum with high probability, and further it is a global minimum with high probability (where high means that the rest probability decreases exponentially in the size of the problem - thus for 10,000 or more nodes, the chances of failure is negligible).

TSP is a touchstone for many general heuristics devised for combinatorial optimization such as genetic algorithms, simulated annealing, Tabu search, Ant colony optimization, and the cross entropy method.

Ant colony optimization

Artificial intelligence researcher Marco Dorigo described in 1997 a method of heuristically generating "good solutions" to the TSP using a simulation of an ant colony called *ACS* (Ant Colony System).[20] It uses some of the same ideas used by real ants to find short paths between food sources and their nest, an emergent behaviour resulting from each ant's preference to follow trail pheromones deposited by other ants.

ACS sends out a large number of virtual ant agents to explore many possible routes on the map. Each ant probabilistically chooses the next city to visit based on a heuristic combining the distance to the city and the amount of virtual pheromone deposited on the edge to the city. The ants explore, depositing pheromone on each edge that they cross, until they have all completed a tour. At this point the ant which completed the shortest tour deposits virtual pheromone along its complete tour route (*global trail updating*). The amount of pheromone deposited is inversely proportional to the tour length; the shorter the tour, the more it deposits.



Special cases

Metric TSP

A very natural restriction of the TSP is to require that the distances between cities form a metric, i.e., they satisfy the triangle inequality. That is, for any 3 cities A, B and C, the distance between A and C must be at most the distance from A to B plus the distance from B to C. Most natural instances of TSP satisfy this constraint.

In this case, there is a constant-factor approximation algorithm due to Christofides[21] that always finds a tour of length at most 1.5 times the shortest tour. In the next paragraphs, we explain a weaker (but simpler) algorithm which finds a tour of length at most twice the shortest tour.

The length of the minimum spanning tree of the network is a natural lower bound for the length of the optimal route. In the TSP with triangle inequality case it is possible to prove upper bounds in terms of the minimum spanning tree and design an algorithm that has a provable upper bound on the length of the route. The first published (and the simplest) example follows.

- Construct the minimum spanning tree.
- Duplicate all its edges. That is, wherever there is an edge from *u* to *v*, add a second edge from *u* to *v*. This gives us an Eulerian graph.
- Find a Eulerian cycle in it. Clearly, its length is twice the length of the tree.
- Convert the Eulerian cycle into the Hamiltonian one in the following way: walk along the Eulerian cycle, and each time you are about to come into an already visited vertex, skip it and try to go to the next one (along the Eulerian cycle).

It is easy to prove that the last step works. Moreover, thanks to the triangle inequality, each skipping at Step 4 is in fact a shortcut, i.e., the length of the cycle does not increase. Hence it gives us a TSP tour no more than twice as long as the optimal one.

The Christofides algorithm follows a similar outline but combines the minimum spanning tree with a solution of another problem, minimum-weight perfect matching. This gives a TSP tour which is at most 1.5 times the optimal. The Christofides algorithm was one of the first approximation algorithms, and was in part responsible for drawing attention to approximation algorithms as a practical approach to intractable problems. As a matter of fact, the term "algorithm" was not commonly extended to approximation algorithms until later; the Christofides algorithm was initially referred to as the Christofides heuristic.

In the special case that distances between cities are all either one or two (and thus the triangle inequality is necessarily satisfied), there is a polynomial-time approximation algorithm that finds a tour of length at most 8/7

times the optimal tour length.[22] However, it is a long-standing (since 1975) open problem to improve the Christofides approximation factor of 1.5 for general metric TSP to a smaller constant. It is known that, unless $P = NP$, there is no polynomial-time algorithm that finds a tour of length at most 220/219=1.00456… times the optimal tour's length.[23] In the case of bounded metrics it is known that there is no polynomial time algorithm that constructs a tour of length at most 321/320 times the optimal tour's length, unless $P = NP$.[24]

Euclidean TSP

Euclidean TSP, or **planar TSP**, is the TSP with the distance being the ordinary Euclidean distance.

Euclidean TSP is a particular case of TSP with triangle inequality, since distances in plane obey triangle inequality. However, it seems to be easier than general TSP with triangle inequality. For example, the minimum spanning tree of the graph associated with an instance of Euclidean TSP is a Euclidean minimum spanning tree, and so can be computed in expected O(n log n) time for n points (considerably less than the number of edges). This enables the simple 2-approximation algorithm for TSP with triangle inequality above to operate more quickly.

In general, for any $c > 0$, there is a polynomial-time algorithm that finds a tour of length at most (1 + 1/c) times the optimal for geometric instances of TSP in O(n (log n)^O(c)) time; this is called a polynomial-time approximation scheme[25] In practice, heuristics with weaker guarantees continue to be used.

Asymmetric TSP

In most cases, the distance between two nodes in the TSP network is the same in both directions. The case where the distance from A to B is not equal to the distance from B to A is called asymmetric TSP. A practical application of an asymmetric TSP is route optimisation using street-level routing (asymmetric due to one-way streets, slip-roads and motorways).

Solving by conversion to Symmetric TSP

Solving an asymmetric TSP graph can be somewhat complex. The following is a 3x3 matrix containing all possible path weights between the nodes A, B and C. One option is to turn an asymmetric matrix of size N into a *symmetric* matrix of size 2N, doubling the complexity.

	A	**B**	**C**
A		1	2
B	6		3
C	5	4	

|+ Asymmetric Path Weights

To double the size, each of the nodes in the graph is duplicated, creating a second *ghost node*. Using duplicate points with very low weights, such as -∞, provides a cheap route "linking" back to the real node and allowing symmetric evaluation to continue. The original 3x3 matrix shown above is visible in the bottom left and the inverse of the original in the top-right. Both copies of the matrix have had their diagonals replaced by the low-cost hop paths, represented by -∞.

	A	B	C	A'	B'	C'
A				-∞	6	5
B				1	-∞	4
C				2	3	-∞
A'	-∞	1	2			
B'	6	-∞	3			
C'	5	4	-∞			

|+ Symmetric Path Weights

The original 3x3 matrix would produce two Hamiltonian cycles (a path that visits every node once), namely A-B-C-A [score 9] and A-C-B-A [score 12]. Evaluating the 6x6 symmetric version of the same problem now produces many paths, including A-A'-B-B'-C-C'-A, A-B'-C-A'-A, A-A'-B-C'-A [all score 9-∞].

The important thing about each new sequence is that there will be an alternation between dashed (A',B',C') and un-dashed nodes (A,B,C) and that the link to "jump" between any related pair (A-A') is effectively free. A version of the algorithm could use any weight for the A-A' path, as long as that weight is *lower* than all other path weights present in the graph. As the path weight to "jump" must effectively be "free", the value zero (0) could be used to represent this cost — if zero is not being used for another purpose already (such as designating invalid paths). In the two examples above, non-existent paths between nodes are shown as a blank square.

Benchmarks

For benchmarking of TSP algorithms, **TSPLIB** [26] is a library of sample instances of the TSP and related problems is maintained, see the TSPLIB external reference. Many of them are lists of actual cities and layouts of actual printed circuits.

Human performance on TSP

The TSP, in particular the Euclidean variant of the problem, has attracted the attention of researchers in cognitive psychology. It is observed that humans are able to produce good quality solutions quickly. The first issue of the Journal of Problem Solving [27] is devoted to the topic of human performance on TSP.

TSP path length for random pointset in a square

Consider N points randomly distributed in a 1 x 1 square with N>>1.

It is known that, for *N* points in a unit square, the TSP always has length at most proportional to the square root of *N*, and that random pointset will have length at least proportional to the square root. However, the constants of proportionality, both for worst-case pointset and for random pointset, are unknown yet.

Lower bound

A lower bound of the shortest tour length is $\frac{1}{2}\sqrt{N}$, obtained by assuming i be a point in the tour sequence and i has its nearest neighbor as its next in the tour.

A better lower bound is $\left(\frac{1}{4}+\frac{3}{8}\right)\sqrt{N}$, obtained by assuming i's next is i's nearest, and i's previous is i's second nearest.

Divide tour sequence into 2 parts as before_i and after_i with each part contains N/2 points. Delete before_i points from the square to build a diluted environment. Now the best next point that i can find in after_i part is $\sqrt{\frac{1}{2N}}$ away. This leads to a lower bound $\sqrt{\frac{N}{2}}$.

- David S. Johnson[28] obtained a lower bound by computer experiment:

$0.7080\sqrt{N} + 0.522$, where 0.522 comes from the points near square boundary which have fewer neighbors.

- Christine L. Valenzuela and Antonia J. Jones [29] obtained another lower bound by computer experiment:

$0.7078\sqrt{N} + 0.551$

Upper bound

By applying simulated annealing method on samples of N=40000, computer shows an upper bound

$\left(\sqrt{\frac{N}{2}} + 0.72\right) \cdot 1.015$, where 0.72 comes from boundary effect.

Because the actual solution is only the shortest path, for the purposes of programmatic search another upper bound is the length of any previously discovered approximation.

See also

- Canadian traveller problem
- Vehicle routing problem
- Route inspection problem
- Set TSP problem
- Seven Bridges of Königsberg
- Traveling repairman problem (minimum latency problem)
- Traveling tourist problem

References

- Applegate, D. L.; Bixby, R. M.; Chvátal, V.; Cook, W. J. (2006), *The Traveling Salesman Problem*, ISBN 0691129932.
- Bellman, R. (1960), "Combinatorial Processes and Dynamic Programming", in Bellman, R., Hall, M., Jr. (eds.), *Combinatorial Analysis, Proceedings of Symposia in Applied Mathematics 10,*, American Mathematical Society, pp. 217–249.
- Bellman, R. (1962), "Dynamic Programming Treatment of the Travelling Salesman Problem", *J. Assoc. Comput. Mach.* **9**: 61–63, doi:10.1145/321105.321111.
- Christofides, N. (1976), *Worst-case analysis of a new heuristic for the travelling salesman problem*, Technical Report 388, Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh.
- Hassin, R.; Rubinstein, S. (2000), "Better approximations for max TSP", *Information Processing Letters* **75**: 181–186, doi:10.1016/S0020-0190(00)00097-1.
- Held, M.; Karp, R. M. (1962), "A Dynamic Programming Approach to Sequencing Problems", *Journal of the Society for Industrial and Applied Mathematics* **10** (1): 196–210, doi:10.1137/0110015.
- Kaplan, H.; Lewenstein, L.; Shafrir, N.; Sviridenko, M. (2004), "Approximation Algorithms for Asymmetric TSP by Decomposing Directed Regular Multigraphs", *In Proc. 44th IEEE Symp. on Foundations of Comput. Sci.*, pp. 56–65.

- Karp, R.M. (1982), "Dynamic programming meets the principle of inclusion and exclusion", *Oper. Res. Lett.* **1**: 49–51, doi:10.1016/0167-6377(82)90044-X.
- Kohn, S.; Gottlieb, A.; Kohn, M. (1977), "A Generating Function Approach to the Traveling Salesman Problem", *ACM Annual Conference*, ACM Press, pp. 294–300.
- Kosaraju, S. R.; Park, J. K.; Stein, C. (1994), "Long tours and short superstrings'", *Proc. 35th Ann. IEEE Symp. on Foundations of Comput. Sci.*, IEEE Computer Society, pp. 166–177.
- Orponen, P.; Mannila, H. (1987), "On approximation preserving reductions: Complete problems and robust measures'", *Technical Report C-1987-28, Department of Computer Science, University of Helsinki*.
- Papadimitriou, C. H.; Yannakakis, M. (1993), "The traveling salesman problem with distances one and two", *Math. Oper. Res.* **18**: 1–11, doi:10.1287/moor.18.1.1.
- Serdyukov, A. I. (1984), "An algorithm with an estimate for the traveling salesman problem of the maximum'", *Upravlyaemye Sistemy* **25**: 80–86.
- Woeginger, G.J. (2003), "Exact Algorithms for NP-Hard Problems: A Survey", *Combinatorial Optimization – Eureka, You Shrink! Lecture notes in computer science, vol. 2570*, Springer, pp. 185–207.

Further reading

- Applegate, D. L.; Bixby, R. E.; Chvátal, V.; Cook, W. J. (2006), *The Traveling Salesman Problem: A Computational Study*, Princeton University Press, ISBN 978-0-691-12993-8.
- Arora, S. (1998), "Polynomial time approximation schemes for Euclidean traveling salesman and other geometric problems" [30], *Journal of the ACM* **45** (5): 753–782, doi:10.1145/290179.290180.
- Babin, Gilbert; Deneault, Stéphanie; Laportey, Gilbert (2005), *Improvements to the Or-opt Heuristic for the Symmetric Traveling Salesman Problem* [31], Cahiers du GERAD, **G-2005-02**, Montreal: Group for Research in Decision Analysis.
- Cook, William; Espinoza, Daniel; Goycoolea, Marcos (2007), "Computing with domino-parity inequalities for the TSP", *INFORMS Journal on Computing* **19** (3): 356–365, doi:10.1287/ijoc.1060.0204.
- Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; Stein, C. (2001), "35.2: The traveling-salesman problem", *Introduction to Algorithms* (2nd ed.), MIT Press and McGraw-Hill, pp. 1027–1033, ISBN 0-262-03293-7.
- Dantzig, G. B.; Fulkerson, and S. M. Johnson, R. (1954), "Solution of a large-scale traveling salesman problem" [32], *Operations Research* **2**: 393–410, doi:10.1287/opre.2.4.393.
- Garey, M. R.; Johnson, D. S. (1979), "A2.3: ND22–24", *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman, pp. 211–212, ISBN 0-7167-1045-5.
- Goldberg, D. E. (1989), *Genetic Algorithms in Search, Optimization & Machine Learning*, New York: Addison-Wesley, ISBN 0201157675.
- Gutin, G.; Yeo, A.; Zverovich, A. (2002), "Traveling salesman should not be greedy: domination analysis of greedy-type heuristics for the TSP", *Discrete Applied Mathematics* **117** (1–3): 81–86, doi:10.1016/S0166-218X(01)00195-0.
- Gutin, G.; Punnen, A. P. (2006), *The Traveling Salesman Problem and Its Variations*, Springer, ISBN 0-387-44459-9.
- Johnson, D. S.; McGeoch, L. A. (1997), "The Traveling Salesman Problem: A Case Study in Local Optimization", in Aarts, E. H. L.; Lenstra, J. K., *Local Search in Combinatorial Optimisation*, John Wiley and Sons Ltd, pp. 215–310.
- Lawler, E. L.; Lenstra, J. K.; Rinnooy Kan, A. H. G.; Shmoys, D. B. (1985), *The Traveling Salesman Problem: A Guided Tour of Combinatorial Optimization*, John Wiley & Sons, ISBN 0-471-90413-9.
- MacGregor, J. N.; Ormerod, T. (1996), "Human performance on the traveling salesman problem" [33], *Perception & Psychophysics* **58** (4): 527–539.
- Mitchell, J. S. B. (1999), "Guillotine subdivisions approximate polygonal subdivisions: A simple polynomial-time approximation scheme for geometric TSP, *k*-MST, and related problems" [34], *SIAM Journal on Computing* **28**:

1298–1309, doi:10.1137/S0097539796309764.
- Rao, S.; Smith, W. (1998), "Approximating geometrical graphs via 'spanners' and 'banyans'", *Proc. 30th Annual ACM Symposium on Theory of Computing*, pp. 540–550.
- Rosenkrantz, Daniel J.; Stearns, Richard E.; Lewis, Philip M., II (1977), "An Analysis of Several Heuristics for the Traveling Salesman Problem", *SIAM Journal on Computing* **6** (5): 563–581, doi:10.1137/0206041.
- Vickers, D.; Butavicius, M.; Lee, M.; Medvedev, A. (2001), "Human performance on visually presented traveling salesman problems", *Psychological Research* **65** (1): 34–45, doi:10.1007/s004260000031, PMID 11505612.
- Walshaw, Chris (2000), *A Multilevel Approach to the Travelling Salesman Problem*, CMS Press.
- Walshaw, Chris (2001), *A Multilevel Lin-Kernighan-Helsgaun Algorithm for the Travelling Salesman Problem*, CMS Press.

External links

- Traveling Salesman Problem [35] at Georgia Tech
- *Traveling Salesman Problem* [36] by Jon McLoone based on a program by Stephen Wolfram, after work by Stan Wagon, Wolfram Demonstrations Project.
- *optimap* [37] an approximation using ACO on GoogleMaps with JavaScript
- Demo applet of a genetic algorithm solving TSPs and VRPTW problems [38]

References

[1] Take one city, and take all possible orders of the other 14 cities. Then divide by two because it does not matter in which direction in time they come after each other: $14!/2 = 43589145600$

[2] "Der Handlungsreisende – wie er sein soll und was er zu thun [sic] hat, um Aufträge zu erhalten und eines glücklichen Erfolgs in seinen Geschäften gewiß zu sein – von einem alten Commis-Voyageur"

[3] A discussion of the early work of Hamilton and Kirkman can be found in Graph Theory 1736-1936

[4] Cited and English translation in Schrijver (2005). Original German: "Wir bezeichnen als *Botenproblem* (weil diese Frage in der Praxis von jedem Postboten, übrigens auch von vielen Reisenden zu lösen ist) die Aufgabe, für endlich viele Punkte, deren paarweise Abstände bekannt sind, den kürzesten die Punkte verbindenden Weg zu finden. Dieses Problem ist natürlich stets durch endlich viele Versuche lösbar. Regeln, welche die Anzahl der Versuche unter die Anzahl der Permutationen der gegebenen Punkte herunterdrücken würden, sind nicht bekannt. Die Regel, man solle vom Ausgangspunkt erst zum nächstgelegenen Punkt, dann zu dem diesem nächstgelegenen Punkt gehen usw., liefert im allgemeinen nicht den kürzesten Weg."

[5] A detailed treatment of the connection between Menger and Whitney as well as the growth in the study of TSP can be found in Alexander Schrijver's 2005 paper "On the history of combinatorial optimization (till 1960). Handbook of Discrete Optimization (K. Aardal, G.L. Nemhauser, R. Weismantel, eds.), Elsevier, Amsterdam, 2005, pp. 1-68. PS (http://homepages.cwi.nl/~lex/files/histco.ps), PDF (http://homepages.cwi.nl/~lex/files/histco.pdf)

[6] http://www.google.com/patents?q=7054798

[7] Behzad, Arash; Modarres, Mohammad (2002), "New Efficient Transformation of the Generalized Traveling Salesman Problem into Traveling Salesman Problem", *Proceedings of the 15th International Conference of Systems Engineering (Las Vegas)*

[8] Orponen (1987)

[9] Papadimitriou (1983)

[10] Christofides (1976)

[11] Kaplan (2004)

[12] Kosaraju (1994)

[13] Serdyukov (1984)

[14] Hassin (2000)

[15] Bellman (1960), Bellman (1962), Held (1962)

[16] Kohn (1977) Karp (1982)

[17] Woeginger (2003)

[18] Work by David Applegate, AT&T Labs - Research, Robert Bixby, ILOG and Rice University, Vašek Chvátal, Concordia University, William Cook, Georgia Tech, and Keld Helsgaun, Roskilde University is discussed on their project web page hosted by Georgia Tech and last updated in June 2004, here (http://www.tsp.gatech.edu/sweden/)

[19] A. B. Kahng and S. Reda, "Match Twice and Stitch: A New TSP Tour Construction Heuristic," Operations Research Letters, 2004, 32(6). pp. 499-509. http://www.sciencedirect.com/science?_ob=GatewayURL&_method=citationSearch&_uoikey=B6V8M-4CKFN5S-4&_origin=SDEMFRASCII&_version=1&md5=04d492ab46c07b9911e230ebecd0f70d

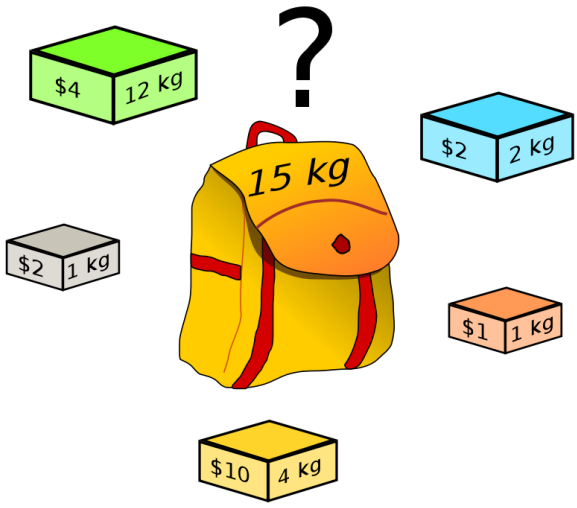
[20] Marco Dorigo. Ant Colonies for the Traveling Salesman Problem. IRIDIA, Université Libre de Bruxelles. IEEE Transactions on Evolutionary Computation, 1(1):53–66. 1997. http://citeseer.ist.psu.edu/86357.html
[21] N. Christofides, *Worst-case analysis of a new heuristic for the traveling salesman problem*, Report 388, Graduate School of Industrial Administration, Carnegie Mellon University, 1976.
[22] P. Berman (2006). M. Karpinski, "8/7-Approximation Algorithm for (1,2)-TSP", Proc. 17th ACM-SIAM SODA (2006), pp. 641-648, ECCC TR05-069 (http://eccc.uni-trier.de/report/2005/069/).
[23] C.H. Papadimitriou and Santosh Vempala. On the approximability of the traveling salesman problem (http://dx.doi.org/10.1007/s00493-006-0008-z), *Combinatorica* 26(1):101–120, 2006.
[24] L. Engebretsen, M. Karpinski, TSP with bounded metrics (http://dx.doi.org/10.1016/j.jcss.2005.12.001). *Journal of Computer and System Sciences*, 72(4):509–546, 2006.
[25] Sanjeev Arora. Polynomial Time Approximation Schemes for Euclidean Traveling Salesman and other Geometric Problems. *Journal of the ACM*, Vol.45, Issue 5, pp.753–782. ISSN:0004-5411. September 1998. http://citeseer.ist.psu.edu/arora96polynomial.html
[26] http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/
[27] http://docs.lib.purdue.edu/jps/
[28] David S. Johnson (http://www.research.att.com/~dsj/papers/HKsoda.pdf)
[29] Christine L. Valenzuela and Antonia J. Jones (http://users.cs.cf.ac.uk/Antonia.J.Jones/Papers/EJORHeldKarp/HeldKarp.pdf)
[30] http://graphics.stanford.edu/courses/cs468-06-winter/Papers/arora-tsp.pdf
[31] http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.89.9953
[32] http://www.jstor.org/stable/166695
[33] http://www.psych.lancs.ac.uk/people/uploads/TomOrmerod20030716T112601.pdf
[34] http://citeseer.ist.psu.edu/622594.html
[35] http://www.tsp.gatech.edu/index.html
[36] http://demonstrations.wolfram.com/TravelingSalesmanProblem/
[37] http://www.gebweb.net/optimap/
[38] http://www.dna-evolutions.com/dnaappletsample.html

Knapsack problem

The **knapsack problem** or **rucksack problem** is a problem in combinatorial optimization: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most useful items.

The problem often arises in resource allocation with financial constraints. A similar problem also appears in combinatorics, complexity theory, cryptography and applied mathematics.

The decision problem form of the knapsack problem is the question "can a value of at least *V* be achieved without exceeding the weight *W*?"



Example of a one-dimensional (constraint) knapsack problem: which boxes should be chosen to maximize the amount of money while still keeping the overall weight under or equal to 15 kg? A multiple constrained problem could consider both the weight and volume of the boxes. Modeling the shapes and sizes would instead constitute a packing problem.(Solution: if any number of each box is available, then three yellow boxes and three grey boxes; if only the shown boxes are available, then all but the green box.)

Definition

In the following, we have *n* kinds of items, 1 through *n*. Each kind of item *i* has a value v_i and a weight w_i. We usually assume that all values and weights are nonnegative. The maximum weight that we can carry in the bag is *W*.

The most common formulation of the problem is the **0-1 knapsack problem**, which restricts the number x_i of copies of each kind of item to zero or one. Mathematically the 0-1-knapsack problem can be formulated as:

- maximize $\sum_{i=1}^{n} v_i x_i$
- subject to $\sum_{i=1}^{n} w_i x_i \leqslant W, \qquad x_i \in \{0,1\}$

The **bounded knapsack problem** restricts the number x_i of copies of each kind of item to a maximum integer value c_i. Mathematically the bounded knapsack problem can be formulated as:

- maximize $\sum_{i=1}^{n} v_i x_i$
- subject to $\sum_{i=1}^{n} w_i x_i \leqslant W, \qquad x_i \in \{0,1,\ldots,c_i\}$

The **unbounded knapsack problem** places no upper bound on the number of copies of each kind of item.

Of particular interest is the special case of the problem with these properties:

- it is a decision problem,
- it is a 0-1 problem,
- for each kind of item, the weight equals the value: $w_i = v_i$.

Notice that in this special case, the problem is equivalent to this: given a set of nonnegative integers, does any subset of it add up to exactly *W*? Or, if negative weights are allowed and *W* is chosen to be zero, the problem is: given a set of integers, does any subset add up to exactly 0? This special case is called the subset sum problem. In the field of cryptography the term *knapsack problem* is often used to refer specifically to the subset sum problem.

If multiple knapsacks are allowed, the problem is better thought of as the bin packing problem.

Computational complexity

The knapsack problem is interesting from the perspective of computer science because

- there is a pseudo-polynomial time algorithm using dynamic programming
- there is a fully polynomial-time approximation scheme, which uses the pseudo-polynomial time algorithm as a subroutine
- the problem is NP-complete to solve exactly, thus it is expected that no algorithm can be both correct and fast (polynomial-time) on all cases
- many cases that arise in practice, and "random instances" from some distributions, can nonetheless be solved exactly.

The subset sum version of the knapsack problem is commonly known as one of Karp's 21 NP-complete problems.

There have been attempts to use subset sum as the basis for public key cryptography systems, such as the Merkle-Hellman knapsack cryptosystem. These attempts typically used some group other than the integers. Merkle-Hellman and several similar algorithms were later broken, because the particular subset sum problems they produced were in fact solvable by polynomial-time algorithms.

One theme in research literature is to identify what the "hard" instances of the knapsack problem look like[1] [2], or viewed another way, to identify what properties of instances in practice might make them more amenable than their worst-case NP-complete behaviour suggests.

Several algorithms are freely available to solve knapsack problems, based on dynamic programming approach[3] , branch and bound approach[4] or hybridations of both approaches[5] [6] [7] [8]

Dynamic programming solution

Unbounded knapsack problem

If all weights ($w_1, \ldots, w_n, W$) are nonnegative integers, the knapsack problem can be solved in pseudo-polynomial time using dynamic programming. The following describes a dynamic programming solution for the *unbounded* knapsack problem.

To simplify things, assume all weights are strictly positive ($w_i > 0$). We wish to maximize total value subject to the constraint that total weight is less than or equal to *W*. Then for each $w \leq W$, define *m[w]* to be the maximum value that can be attained with total weight less than or equal to *w*. *m[W]* then is the solution to the problem.

Observe that *m[w]* has the following properties:

- $m[0] = 0$(the sum of zero items, i.e., the summation of the empty set)
- $m[w] = \max(m[w-1], \max_{w_i \leq w}(v_i + m[w - w_i]))$

where v_i is the value of the *i*-th kind of item.

Here the maximum of the empty set is taken to be zero. Tabulating the results from $m[0]$up through $m[W]$gives the solution. Since the calculation of each $m[w]$involves examining n items, and there are W values of $m[w]$to calculate, the running time of the dynamic programming solution is $O(nW)$. Dividing $w_1, w_2, \ldots, w_n, W$ by their greatest common divisor is an obvious way to improve the running time.

The $O(nW)$complexity does not contradict the fact that the knapsack problem is NP-complete, since W, unlike n, is not polynomial in the length of the input to the problem. The length of the input to the problem is proportional to the number, $\log W$, of bits in W, not to W itself.

0-1 knapsack problem

A similar dynamic programming solution for the *0-1 knapsack problem* also runs in pseudo-polynomial time. As above, assume $w_1, w_2, \ldots, w_n, W$ are strictly positive integers. Define $m[i, w]$to be the maximum value that can be attained with weight less than or equal to w using items up to i .

We can define $m[i, w]$recursively as follows:

- $m[0, w] = 0$
- $m[i, 0] = 0$
- $m[i, w] = m[i-1, w]$; if $w_i > w$
- $m[i, w] = \max(m[i-1, w], m[i-1, w-w_i] + v_i)$; if $w_i \leqslant w$.

The solution can then be found by calculating $m[n, W]$. To do this efficiently we can use a table to store previous computations. This solution will therefore run in $O(nW)$time and $O(nW)$space.

Greedy approximation algorithm

George Dantzig proposed (1957) a greedy approximation algorithm to solve the unbounded knapsack problem. His version sorts the items in decreasing order of value per unit of weight, v_i/w_i. It then proceeds to insert them into the sack, starting with as many copies as possible of the first kind of item until there is no longer space in the sack for more. Provided that there is an unlimited supply of each kind of item, if m is the maximum value of items that fit into the sack, then the greedy algorithm is guaranteed to achieve at least a value of $m/2$. However, for the bounded problem, where the supply of each kind of item is limited, the algorithm may be far from optimal.

Dominance relations to simplify the resolution of the unbounded knapsack problem

Some relations between items are such that quite a lot of items may be useless to consider to build an optimal solution. These relations are known as *Dominance relations*. When an item "i" is known to be dominated by a set of items "J", it can be thrown out of the set of items usable to build an optimal value. The dominance relations between items allow the size of the search space to be significantly reduced. All the dominance relations, enumerated below, could be derived by the following inequalities: $\sum_{j \in J} w_j \, x_j \ \leq \alpha \, w_i$, and $\sum_{j \in J} p_j \, x_j \ \geq \alpha \, p_i$ for some $x \in Z_+^n$ where $\alpha \in Z_+$, $J \subseteq N \, i \notin J$

Collective dominance

The *i*-th item is **collectively dominated** by *J*, written as $i \ll J$ iff $\sum_{j \in J} w_j \, x_j \ \leq w_i$ and $\sum_{j \in J} p_j \, x_j \ \geq p_i$ for some $x \in Z_+^n$ i.e. $\alpha = 1$. The verification of this dominance is computationally hard, so it can be used in a dynamic programming approach only.

Threshold dominance

the *i*-th item is **threshold dominated** by *J*, written as $i \prec\prec J$ iff (the above inequalities hold when $\alpha \geq 1$. This is an obvious generalization of the collective dominance by using instead of single item "i" a compound one, say α times item "i". The smallest such α defines the **threshold** of the item "i", written $t_i = (\alpha - 1) w_i$.

Multiple dominance

The item "i" is **multiply dominated** by "j", written as $i \ll_m j$, iff $w_j \, x_j \ \leq w_i$, and $p_j \, x_j \ \geq p_i$ for some $x_j \in Z_+$ i.e. $J = \{j\}, \alpha = 1, x_j = \lfloor \frac{w_i}{w_j} \rfloor$. This dominance could be efficiently used in a preprocessing because it can be detected relatively easily.

Modular dominance

Let b = the *best item*, i.e $\frac{p_b}{w_b} \geq \frac{p_j}{w_j}$ for all *j* The item *i* is **modularly dominated** by *j*, written as $i \ll_{\equiv} j$ iff $w_j = w_i + t w_b, t \leq 0$, and $p_j - t p_b \geq p_i$ i.e. $J = \{b, j\}, \alpha = 1, x_b = -t, x_j = 1$

See also

- List of knapsack problems
- Packing problem
- Cutting stock problem
- Continuous knapsack problem

Further reading

- Garey, Michael R.; David S. Johnson (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman. ISBN 0-7167-1045-5. A6: MP9, pg.247.
- Martello, Silvano; Paolo Toth (1990). *Knapsack Problems: Algorithms and Computer Implementations* [9]. John Wiley & Sons. ISBN 0-471-92420-2.
- Kellerer, Hans; U. Pferschy, D. Pisinger (2005). *Knapsack Problems*. Springer Verlag. ISBN 3-540-40286-1.

- inria-00335065, version 1 -, 00335065 (28 Oct 2008). http://hal.inria.fr/docs/00/33/50/65/PDF/jvers08.pdf: version 1.

External links

- Lecture slides on the knapsack problem [10]
- PYAsUKP: Yet Another solver for the Unbounded Knapsack Problem [11], with code, benchmarks and downloadable copies of some papers.
- Home page of David Pisinger [12] with downloadable copies of some papers on the publication list (including "Where are the hard knapsack problems?")
- Knapsack Problem solutions in many languages [13] at Rosetta Code
- Dynamic Programming algorithm to 0/1 Knapsack problem [14]
- 0-1 Knapsack Problem in Python [15]
- Interactive JavaScript branch-and-bound solver [16]

References

[1] Pisinger, D. 2003. Where are the hard knapsack problems? Technical Report 2003/08, Department of Computer Science, University of Copenhagen, Copenhagen, Denmark.
[2] L. Caccetta, A. Kulanoot, Computational Aspects of Hard Knapsack Problems, Nonlinear Analysis 47 (2001) 5547–5558.
[3] Rumen Andonov, Vincent Poirriez, Sanjay Rajopadhye (2000) Unbounded Knapsack Problem : dynamic programming revisited European Journal of Operational Research 123: 2. 168-181 http://dx.doi.org/10.1016/S0377-2217(99)00265-9
[4] S. Martello, P. Toth, Knapsack Problems: Algorithms and Computer Implementation , John Wiley and Sons, 1990
[5] S. Martello, D. Pisinger, P. Toth, Dynamic programming and strong bounds for the 0-1 knapsack problem , Manag. Sci., 45:414-424, 1999.
[6] Vincent Poirriez, Nicola Yanev, Rumen Andonov (2009) A Hybrid Algorithm for the Unbounded Knapsack Problem Discrete Optimization http://dx.doi.org/10.1016/j.disopt.2008.09.004
[7] G. Plateau, M. Elkihel, A hybrid algorithm for the 0-1 knapsack problem, Methods of Oper. Res., 49:277-293, 1985.
[8] S. Martello, P. Toth, A mixture of dynamic programming and branch-and-bound for the subset-sum problem, Manag. Sci., 30:765-771
[9] http://www.or.deis.unibo.it/knapsack.html
[10] http://www.cse.unl.edu/~goddard/Courses/CSCE310J/Lectures/Lecture8-DynamicProgramming.pdf
[11] http://download.gna.org/pyasukp
[12] http://www.diku.dk/~pisinger/
[13] http://www.rosettacode.org/wiki/Knapsack_Problem
[14] http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Dynamic/knapsackdyn.htm
[15] http://sites.google.com/site/mikescoderama/Home/0-1-knapsack-problem-in-p
[16] http://allievi.sssup.it/jacopo/BB/

Nonlinear programming

In mathematics, **nonlinear programming** (**NLP**) is the process of solving a system of equalities and inequalities, collectively termed constraints, over a set of unknown real variables, along with an objective function to be maximized or minimized, where some of the constraints or the objective function are nonlinear.

Applicability

A typical nonconvex problem is that of optimizing transportation costs by selection from a set of transportion methods, one or more of which exhibit economies of scale, with various connectivities and capacity constraints. An example would be petroleum product transport given a selection or combination of pipeline, rail tanker, road tanker, river barge, or coastal tankship. Owing to economic batch size the cost functions may have discontinuities in addition to smooth changes.

Mathematical formulation of the problem

The problem can be stated simply as:

$\max_{x \in X} f(x)$ to maximize some variable such as product throughput

or

$\min_{x \in X} f(x)$ to minimize a cost function

where

$f : R^n \to R$

$X \subseteq R^n.$

Methods for solving the problem

If the objective function *f* is linear and the constrained space is a polytope, the problem is a linear programming problem, which may be solved using well known linear programming solutions.

If the objective function is concave (maximization problem), or convex (minimization problem) and the constraint set is convex, then the program is called convex and general methods from convex optimization can be used.

Several methods are available for solving nonconvex problems. One approach is to use special formulations of linear programming problems. Another method involves the use of branch and bound techniques, where the program is divided into subclasses to be solved with convex (minimization problem) or linear approximations that form a lower bound on the overall cost within the subdivision. With subsequent divisions, at some point an actual solution will be obtained whose cost is equal to the best lower bound obtained for any of the approximate solutions. This solution is optimal, although possibly not unique. The algorithm may also be stopped early, with the assurance that the best possible solution is within a tolerance from the best point found; such points are called ε-optimal. Terminating to ε-optimal points is typically necessary to ensure finite termination. This is especially useful for large, difficult problems and problems with uncertain costs or values where the uncertainty can be estimated with an appropriate reliability estimation.

Under differentiability and constraint qualifications, the Karush-Kuhn-Tucker (KKT) conditions provide necessary conditions for a solution to be optimal. Under convexity, these conditions are also sufficient.

Examples

2-dimensional example

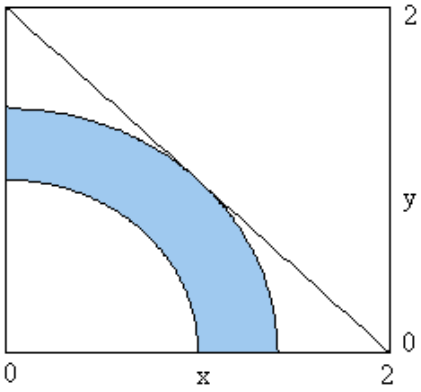
A simple problem can be defined by the constraints

$$x_1 \geq 0$$
$$x_2 \geq 0$$
$$x_1^2 + x_2^2 \geq 1$$
$$x_1^2 + x_2^2 \leq 2$$

with an objective function to be maximized

$$f(\mathbf{x}) = x_1 + x_2$$

where $\mathbf{x} = (x_1, x_2)$.



The intersection of the line with the constrained space represents the solution

3-dimensional example

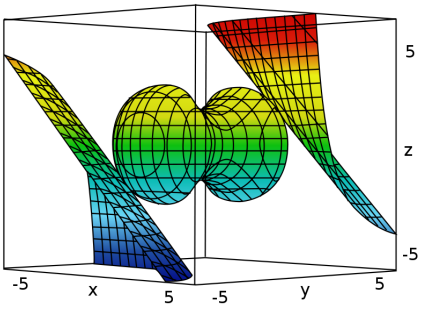
Another simple problem can be defined by the constraints

$$x_1^2 - x_2^2 + x_3^2 \leq 2$$
$$x_1^2 + x_2^2 + x_3^2 \leq 10$$

with an objective function to be maximized

$$f(\mathbf{x}) = x_1 x_2 + x_2 x_3$$

where $\mathbf{x} = (x_1, x_2, x_3)$.



The intersection of the top surface with the constrained space in the center represents the solution

See also

- Curve fitting

- Least squares minimization
- Linear programming
- nl (format)
- Optimization (mathematics)

Further reading

- Avriel, Mordecai (2003). *Nonlinear Programming: Analysis and Methods.* Dover Publishing. ISBN 0-486-43227-0.
- Bazaraa, Mokhtar S. and Shetty, C. M. (1979). *Nonlinear programming. Theory and algorithms.* John Wiley & Sons. ISBN 0-471-78610-1.
- Bertsekas, Dimitri P. (1999). *Nonlinear Programming: 2nd Edition.* Athena Scientific. ISBN 1-886529-00-0.
- Luenberger, David G.; Ye, Yinyu (2008). *Linear and nonlinear programming*. International Series in Operations Research & Management Science. **116** (Third ed.). New York: Springer. pp. xiv+546. ISBN 978-0-387-74502-2. MR2423726
- Nocedal, Jorge and Wright, Stephen J. (1999). *Numerical Optimization.* Springer. ISBN 0-387-98793-2.

External links

- Nonlinear programming FAQ [1]
- Mathematical Programming Glossary [2]
- Nonlinear Programming Survey OR/MS Today [3]

References

'Optimization: Insights and Applications', Jan Brinkhuis and Vladimir Tikhomirov: 2005, Princeton University Press

References

[1] http://www-unix.mcs.anl.gov/otc/Guide/faq/nonlinear-programming-faq.html

[2] http://glossary.computing.society.informs.org/

[3] http://www.lionhrtpub.com/orms/surveys/nlp/nlp.html

Assignment problem

The **assignment problem** is one of the fundamental combinatorial optimization problems in the branch of optimization or operations research in mathematics. It consists of finding a maximum weight matching in a weighted bipartite graph.

In its most general form, the problem is as follows:

> There are a number of *agents* and a number of *tasks*. Any agent can be assigned to perform any task, incurring some *cost* that may vary depending on the agent-task assignment. It is required to perform all tasks by assigning exactly one agent to each task in such a way that the *total cost* of the assignment is minimized.

If the numbers of agents and tasks are equal and the total cost of the assignment for all tasks is equal to the sum of the costs for each agent (or the sum of the costs for each task, which is the same thing in this case), then the problem is called the *Linear assignment problem*. Commonly, when speaking of the *Assignment problem* without any additional qualification, then the *Linear assignment problem* is meant.

Algorithms and generalizations

The Hungarian algorithm is one of many algorithms that have been devised that solve the linear assignment problem within time bounded by a polynomial expression of the number of agents.

The assignment problem is a special case of the transportation problem, which is a special case of the minimum cost flow problem, which in turn is a special case of a linear program. While it is possible to solve any of these problems using the simplex algorithm, each specialization has more efficient algorithms designed to take advantage of its special structure. If the cost function involves quadratic inequalities it is called the quadratic assignment problem.

Example

Suppose that a taxi firm has three taxis (the agents) available, and three customers (the tasks) wishing to be picked up as soon as possible. The firm prides itself on speedy pickups, so for each taxi the "cost" of picking up a particular customer will depend on the time taken for the taxi to reach the pickup point. The solution to the assignment problem will be whichever combination of taxis and customers results in the least total cost.

However, the assignment problem can be made rather more flexible than it first appears. In the above example, suppose that there are four taxis available, but still only three customers. Then a fourth dummy task can be invented, perhaps called "sitting still doing nothing", with a cost of 0 for the taxi assigned to it. The assignment problem can then be solved in the usual way and still give the best solution to the problem.

Similar tricks can be played in order to allow more tasks than agents, tasks to which multiple agents must be assigned (for instance, a group of more customers than will fit in one taxi), or maximizing profit rather than minimizing cost.

Formal mathematical definition

The formal definition of the **assignment problem** (or **linear assignment problem**) is

> Given two sets, A and T, of equal size, together with a weight function $C : A \times T \to \mathbf{R}$. Find a bijection $f : A \to T$ such that the cost function:
>
> $$\sum_{a \in A} C(a, f(a))$$
>
> is minimized.

Usually the weight function is viewed as a square real-valued matrix C, so that the cost function is written down as:

$$\sum_{a \in A} C_{a,f(a)}$$

The problem is "linear" because the cost function to be optimized as well as all the constraints contain only linear terms.

The problem can be expressed as a standard linear program with the objective function

$$\sum_{i \in A} \sum_{j \in T} C(i,j) x_{ij}$$

subject to the constraints

$$\sum_{j \in T} x_{ij} = 1 \text{for } i \in A\,,$$

$$\sum_{i \in A} x_{ij} = 1 \text{for } j \in T,$$

$$x_{ij} \geq 0 \text{for } i, j \in A, T.$$

The variable x_{ij} represents the assignment of agent i to task j, taking value 1 if the assignment is done and 0 otherwise. This formulation allows also fractional variable values, but there is always an optimal solution where the variables take integer values. This is because the constraint matrix is totally unimodular. The first constraint requires that every agent is assigned to exactly one task, and the second constraint requires that every task is assigned exactly one agent.

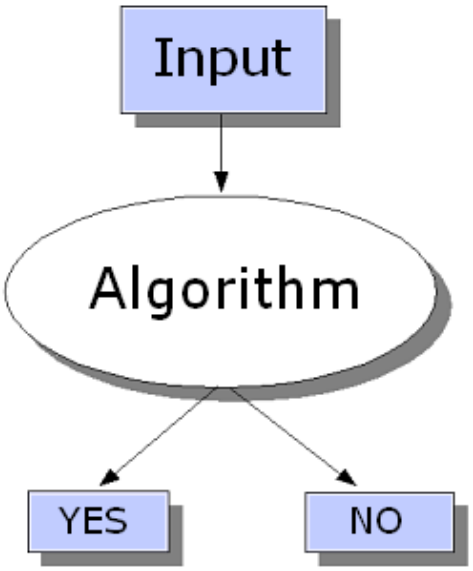
See also

- Stable marriage problem
- Auction algorithm
- Generalized assignment problem

Further reading

- Burkard, Rainer; M. Dell'Amico, S. Martello (2009). *Assignment Problems*. SIAM. ISBN 978-0-898716-63-4.

Decision problem



A **decision problem** has only two possible outputs, *yes* or *no* (or alternately 1 or 0) on any input.

In computability theory and computational complexity theory, a **decision problem** is a question in some formal system with a yes-or-no answer, depending on the values of some input parameters. For example, the problem "given two numbers *x* and *y*, does *x* evenly divide *y*?" is a decision problem. The answer can be either 'yes' or 'no', and depends upon the values of *x* and *y*.

Decision problems are closely related to function problems, which can have answers that are more complex than a simple 'yes' or 'no'. A corresponding function problem is "given two numbers *x* and *y*, what is *x* divided by *y*?". They are also related to optimization problems, which are concerned with finding the *best* answer to a particular problem.

A method for solving a decision problem given in the form of an algorithm is called a **decision procedure** for that problem. A decision procedure for the decision problem "given two numbers *x* and *y*, does *x* evenly divide *y*?" would give the steps for determining whether *x* evenly divides *y*, given *x* and *y*. One such algorithm is long division, taught to many school children. If the remainder is zero the answer produced is 'yes', otherwise it is 'no'. A decision problem which can be solved by an algorithm, such as this example, is called **decidable**.

The field of computational complexity categorizes *decidable* decision problems by how difficult they are to solve. "Difficult", in this sense, is described in terms of the computational resources needed by the most efficient algorithm for a certain problem. The field of recursion theory, meanwhile, categorizes *undecidable* decision problems by Turing degree, which is a measure of the noncomputability inherent in any solution.

Research in computability theory has typically focused on decision problems. As explained in the section Equivalence with function problems below, there is no loss of generality.

Definition

A *decision problem* is any arbitrary yes-or-no question on an infinite set of inputs. Because of this, it is traditional to define the decision problem equivalently as: the set of inputs for which the problem returns *yes*.

These inputs can be natural numbers, but also other values of some other kind, such as strings of a formal language. Using some encoding, such as Gödel numberings, the strings can be encoded as natural numbers. Thus, a decision problem informally phrased in terms of a formal language is also equivalent to a set of natural numbers. To keep the formal definition simple, it is phrased in terms of subsets of the natural numbers.

Formally, a **decision problem** is a subset of the natural numbers. The corresponding informal problem is that of deciding whether a given number is in the set.

Examples

A classic example of a decidable decision problem is the set of prime numbers. It is possible to effectively decide whether a given natural number is prime by testing every possible nontrivial factor. Although much more efficient methods of primality testing are known, the existence of any effective method is enough to establish decidability.

Decidability

A decision problem *A* is called **decidable** or **effectively solvable** if *A* is a recursive set. A problem is called **partially decidable**, **semidecidable**, **solvable**, or **provable** if *A* is a recursively enumerable set. Partially decidable problems and any other problems that are not decidable are called **undecidable**.

The halting problem is an important undecidable decision problem; for more examples, see list of undecidable problems.

Complete problems

Decision problems can be ordered according to many-one reducibility and related feasible reductions such as Polynomial-time reductions. A decision problem *P* is said to be **complete** for a set of decision problems *S* if *P* is a member of *S* and every problem in *S* can be reduced to *P*. Complete decision problems are used in computational complexity to characterize complexity classes of decision problems. For example, the Boolean satisfiability problem is complete for the class NP of decision problems under polynomial-time reducibility.

History

The *Entscheidungsproblem*, German for "Decision-problem", is attributed to David Hilbert: "At [the] 1928 conference Hilbert made his questions quite precise. First, was mathematics *complete*... Second, was mathematics *consistent*... And thirdly, was mathematics *decidable*? By this he meant, did there exist a definite method which could, in principle be applied to any assertion, and which was guaranteed to produce a correct decision on whether that assertion was true" (Hodges, p. 91). Hilbert believed that "in mathematics there is no ignorabimus' (Hodges, p. 91ff) meaning 'we will not know'. See David Hilbert and Halting Problem for more.

Equivalence with function problems

A function problem consists of a partial function *f*; the informal "problem" is to compute the values of *f* on the inputs for which it is defined.

Every function problem can be turned into a decision problem; the decision problem is just the graph of the associated function. (The graph of a function *f* is the set of pairs (x,y) such that $f(x) = y$.) If this decision problem were effectively solvable then the function problem would be as well. This reduction does not respect computational complexity, however. For example, it is possible for the graph of a function to be decidable in polynomial time (in which case running time is computed as a function of the pair (x,y)) when the function is not computable in polynomial time (in which case running time is computed as a function of x alone). The function $f(x) = 2^x$ has this property.

Every decision problem can be converted into the function problem of computing the characteristic function of the set associated to the decision problem. If this function is computable then the associated decision problem is decidable. However, this reduction is more liberal than the standard reduction used in computational complexity (sometimes called polynomial-time many-one reduction); for example, the complexity of the characteristic functions of an NP-complete problem and its co-NP-complete complement is exactly the same even though the underlying decision problems may not be considered equivalent in some typical models of computation.

Practical decision

Having practical decision procedures for classes of logical formulas is of considerable interest for program verification and circuit verification. Pure Boolean logical formulas are usually decided using SAT-solving techniques based on the DPLL algorithm. Conjunctive formulas over linear real or rational arithmetic can be decided using the Simplex algorithm, formulas in linear integer arithmetic (Presburger arithmetic) can be decided using Cooper's algorithm or William Pugh's Omega test. Formulas with negations, conjunctions and disjunctions combine the difficulties of satisfiability testing with that of decision of conjunctions; they are generally decided nowadays using SMT-solving technique, which combine SAT-solving with decision procedures for conjunctions and propagation techniques. Real polynomial arithmetic, also known as the theory of real closed fields, is decidable, for instance using the Cylindrical algebraic decomposition; unfortunately the complexity of that algorithm is excessive for most practical uses.

A leading scientific conference in this field is CAV.

See also

- ALL (complexity)
- Decidability (logic) – for the problem of deciding whether a formula is a consequence of a logical theory.
- yes-no question
- Optimization problem
- Search problem
- Counting problem (complexity)
- Function problem

References

- Hanika, Jiri. *Search Problems and Bounded Arithmetic.* PhD Thesis, Charles University, Prague. http://www.eccc.uni-trier.de/static/books/Search_Problems_and_Bounded_Arithmetic/
- Hodges, A., *Alan Turing: The Enigma*, Simon and Schuster, New York. Cf Chapter "The Spirit of Truth" for some more history that led to Turing's work.

  Hodges references a biography of David Hilbert: Constance Reid, *Hilbert* (George Allen & Unwin; Springer-Verlag, 1970). There are apparently more recent editions.
- Kozen, D.C. (1997), *Automata and Computability*, Springer.
- Hartley Rogers, Jr., *The Theory of Recursive Functions and Effective Computability*, MIT Press, ISBN 0-262-68052-1 (paperback), ISBN 0-07-053522-1
- Sipser, M. (1996), *Introduction to the Theory of Computation*, PWS Publishing Co.
- Robert I. Soare (1987), *Recursively Enumerable Sets and Degrees*, Springer-Verlag, ISBN 0-387-15299-7

Effective decision

- Daniel Kroening & Ofer Strichman, *Decision procedures*, Springer, ISBN 978-3-540-74104-6
- Aaron Bradley & Zohar Manna, *The calculus of computation*, Springer, ISBN 978-3-540-74112-1

Proof theory

Proof theory is a branch of mathematical logic that represents proofs as formal mathematical objects, facilitating their analysis by mathematical techniques. Proofs are typically presented as inductively-defined data structures such as plain lists, boxed lists, or trees, which are constructed according to the axioms and rules of inference of the logical system. As such, proof theory is syntactic in nature, in contrast to model theory, which is semantic in nature. Together with model theory, axiomatic set theory, and recursion theory, proof theory is one of the so-called *four pillars* of the foundations of mathematics.[1]

Proof theory is important in philosophical logic, where the primary interest is in the idea of a proof-theoretic semantics, an idea which depends upon technical ideas in structural proof theory to be feasible.

History

Although the formalisation of logic was much advanced by the work of such figures as Gottlob Frege, Giuseppe Peano, Bertrand Russell, and Richard Dedekind, the story of modern proof theory is often seen as being established by David Hilbert, who initiated what is called Hilbert's program in the foundations of mathematics. Kurt Gödel's seminal work on proof theory first advanced, then refuted this program: his completeness theorem initially seemed to bode well for Hilbert's aim of reducing all mathematics to a finitist formal system; then his incompleteness theorems showed that this is unattainable. All of this work was carried out with the proof calculi called the Hilbert systems.

In parallel, the foundations of structural proof theory were being founded. Jan Łukasiewicz suggested in 1926 that one could improve on Hilbert systems as a basis for the axiomatic presentation of logic if one allowed the drawing of conclusions from assumptions in the inference rules of the logic. In response to this Stanisław Jaśkowski (1929) and Gerhard Gentzen (1934) independently provided such systems, called calculi of natural deduction, with Gentzen's approach introducing the idea of symmetry between the grounds for asserting propositions, expressed in introduction rules, and the consequences of accepting propositions in the elimination rules, an idea that has proved very important in proof theory[2] . Gentzen (1934) further introduced the idea of the sequent calculus, a calculus advanced in a similar spirit that better expressed the duality of the logical connectives[3] , and went on to make fundamental advances in the formalisation of intuitionistic logic, and provide the first combinatorial proof of the consistency of Peano arithmetic. Together, the presentation of natural deduction and the sequent calculus introduced the fundamental idea of analytic proof to proof theory,

Formal and informal proof

The *informal* proofs of everyday mathematical practice are unlike the *formal* proofs of proof theory. They are rather like high-level sketches that would allow an expert to reconstruct a formal proof at least in principle, given enough time and patience. For most mathematicians, writing a fully formal proof is too pedantic and long-winded to be in common use.

Formal proofs are constructed with the help of computers in interactive theorem proving. Significantly, these proofs can be checked automatically, also by computer. (Checking formal proofs is usually simple, whereas *finding* proofs (automated theorem proving) is generally hard.) An informal proof in the mathematics literature, by contrast, requires weeks of peer review to be checked, and may still contain errors.

Kinds of proof calculi

The three most well-known styles of proof calculi are:

- The Hilbert calculi
- The natural deduction calculi
- The sequent calculi

Each of these can give a complete and axiomatic formalization of propositional or predicate logic of either the classical or intuitionistic flavour, almost any modal logic, and many substructural logics, such as relevance logic or linear logic. Indeed it is unusual to find a logic that resists being represented in one of these calculi.

Consistency proofs

As previously mentioned, the spur for the mathematical investigation of proofs in formal theories was Hilbert's program. The central idea of this program was that if we could give finitary proofs of consistency for all the sophisticated formal theories needed by mathematicians, then we could ground these theories by means of a metamathematical argument, which shows that all of their purely universal assertions (more technically their provable $\prod_1^0$sentences) are finitarily true; once so grounded we do not care about the non-finitary meaning of their existential theorems, regarding these as pseudo-meaningful stipulations of the existence of ideal entities.

The failure of the program was induced by Kurt Gödel's incompleteness theorems, which showed that any ω-consistent theory that is sufficiently strong to express certain simple arithmetic truths, cannot prove its own consistency, which on Gödel's formulation is a $\prod_1^0$sentence.

Much investigation has been carried out on this topic since, which has in particular led to:

- Refinement of Gödel's result, particularly J. Barkley Rosser's refinement, weakening the above requirement of ω-consistency to simple consistency;
- Axiomatisation of the core of Gödel's result in terms of a modal language, provability logic;
- Transfinite iteration of theories, due to Alan Turing and Solomon Feferman;
- The recent discovery of self-verifying theories, systems strong enough to talk about themselves, but too weak to carry out the diagonal argument that is the key to Gödel's unprovability argument.

See also Mathematical logic

Structural proof theory

Structural proof theory is the subdiscipline of proof theory that studies proof calculi that support a notion of analytic proof. The notion of analytic proof was introduced by Gentzen for the sequent calculus; there the analytic proofs are those that are cut-free. His natural deduction calculus also supports a notion of analytic proof, as shown by Dag Prawitz. The definition is slightly more complex: we say the analytic proofs are the normal forms, which are related to the notion of normal form in term rewriting. More exotic proof calculi such as Jean-Yves Girard's proof nets also support a notion of analytic proof.

Structural proof theory is connected to type theory by means of the Curry-Howard correspondence, which observes a structural analogy between the process of normalisation in the natural deduction calculus and beta reduction in the typed lambda calculus. This provides the foundation for the intuitionistic type theory developed by Per Martin-Löf, and is often extended to a three way correspondence, the third leg of which are the cartesian closed categories.

Proof-theoretic semantics

In linguistics, type-logical grammar, categorial grammar and Montague grammar apply formalisms based on structural proof theory to give a formal natural language semantics.

Tableau systems

Analytic tableaux apply the central idea of analytic proof from structural proof theory to provide decision procedures and semi-decision procedures for a wide range of logics.

Ordinal analysis

Ordinal analysis is a powerful technique for providing combinatorial consistency proofs for theories formalising arithmetic and analysis.

Logics from proof analysis

Several important logics have come from insights into logical structure arising in structural proof theory.

See also

- Proof techniques
- Intermediate logics

References

- J. Avigad, E.H. Reck (2001). "Clarifying the nature of the infinite": the development of metamathematics and proof theory [4]. Carnegie-Mellon Technical Report CMU-PHIL-120.
- J. Barwise (ed., 1978). Handbook of Mathematical Logic. North-Holland.
- 2πix.com: Logic [5] Part of a series of articles covering mathematics and logic.
- A. S. Troelstra, H. Schwichtenberg (1996). *Basic Proof Theory*. In series *Cambridge Tracts in Theoretical Computer Science*, Cambridge University Press, ISBN 0-521-77911-1.
- G. Gentzen (1935/1969). Investigations into logical deduction. In M. E. Szabo, editor, *Collected Papers of Gerhard Gentzen*. North-Holland. Translated by Szabo from "Untersuchungen über das logische Schliessen", Mathematisches Zeitschrift 39: 176-210, 405-431.
- L. Moreno-Armella & B.Sriraman (2005).*Structural Stability and Dynamic Geometry: Some Ideas on Situated Proof. International Reviews on Mathematical Education. Vol. 37, no.3, pp.130-139* [6]
- J. von Plato (2008). The Development of Proof Theory [7]. Stanford Encyclopedia of Philosophy.
- Wang, Hao (1981). *Popular Lectures on Mathematical Logic*. Van Nostrand Reinhold Company. ISBN 0442231091.

References

[1] E.g., Wang (1981), pp. 3–4, and Barwise (1978).
[2] Prawitz (1965).
[3] Girard, Lafont, and Taylor (1988).
[4] http://www.andrew.cmu.edu/user/avigad/Papers/infinite.pdf
[5] http://2piix.com/articles/title/Logic/
[6] http://www.springerlink.com/content/n602313107541846/?p=74ab8879ce75445da488d5744cbc3818&pi=0
[7] http://plato.stanford.edu/entries/proof-theory-development/

Optimization problem

In mathematics and computer science, an **optimization problem** is the problem of finding the *best* solution from all feasible solutions. More formally, an optimization problem A is a quadruple (I, f, m, g), where

- I is a set of instances;
- given an instance $x \in I$, $f(x)$ is the set of feasible solutions;
- given an instance x and a feasible solution y of x, $m(x, y)$ denotes the measure of y, which is usually a positive real.
- g is the goal function, and is either $\min$ or $\max$.

The goal is then to find for some instance x an *optimal solution*, that is, a feasible solution y with

$$m(x, y) = g\{m(x, y') \mid y' \in f(x)\}.$$

For each optimization problem, there is a corresponding decision problem that asks whether there is a feasible solution for some particular measure m_0. For example, if there is a graph G which contains vertices u and v, an optimization problem might be "find a path from u to v that uses the fewest edges". This problem might have an answer of, say, 4. A corresponding decision problem would be "is there a path from u to v that uses 10 or fewer edges?" This problem can be answered with a simple 'yes' or 'no'.

In the field of approximation algorithms, algorithms are designed to find near-optimal solutions to hard problems. The usual decision version is then an inadequate definition of the problem since it only specifies acceptable solutions. Even though we could introduce suitable decision problems, the problem is more naturally characterized as an optimization problem.[1]

NP optimization problems

An *NP-optimization problem* (NPO) is an optimization problem with the following additional conditions.[2] Note that the below referred polynomials are functions of the size of the respective functions' inputs, not the size of some implicit set of input instances.

- the size of every feasible solution $y \in f(x)$ is polynomially bounded in the size of the given instance x,
- the languages $\{ x \mid x \in I \}$ and $\{ (x,y) \mid y \in f(x) \}$ can be recognized in polynomial time, and
- *m* is polynomial-time computable.

This implies that the corresponding decision problem is in NP. In computer science, interesting optimization problems usually have the above properties and are therefore NPO problems. A problem is additionally called a P-optimization (PO) problem, if there exists an algorithm which finds optimal solutions in polynomial time. Often, when dealing with the class NPO, one is interested in optimization problems for which the decision versions are NP-hard. Note that hardness relations are always with respect to some reduction. Due to the connection between approximation algorithms and computational optimization problems, reductions which preserve approximation in some respect are for this subject preferred than the usual Turing and Karp reductions. An example of such a reduction would be the L-reduction. For this reason, optimization problems with NP-complete decision versions are not necessarily called NPO-complete.[3]

NPO is divided into the following subclasses according to their approximability:[2]

- *NPO(I)*: Equals FPTAS. Contains the Knapsack problem.
- *NPO(II)*: Equals PTAS. Contains the Makespan scheduling problem.
- *NPO(III)*: :The class of NPO problems that have polynomial-time algorithms which computes solutions with a cost at most *c* times the optimal cost (for minimization problems) or a cost at least $1/c$ of the optimal cost (for maximization problems). In Hromkovic's book, excluded from this class are all NPO(II)-problems save if P=NP. Without the exclusion, equals APX. Contains MAX-SAT and metric TSP.
- *NPO(IV)*: :The class of NPO problems with polynomial-time algorithms approximating the optimal solution by a ratio that is polynomial in a logarithm of the size of the input. In Hromkovic's book, excluded from this class are all NPO(III)-problems save if P=NP. Contains the set cover problem.
- *NPO(V)*: :The class of NPO problems with polynomial-time algorithms approximating the optimal solution by a ratio bounded by some function on n. In Hromkovic's book, excluded from this class are all NPO(IV)-problems save if P=NP. Contains the TSP and Max Clique problems.

Another class of interest is NPOPB, NPO with polynomially bounded cost functions. Problems with this condition have many desirable properties.

See also

- Optimization (mathematics)
- Semi-infinite programming
- Decision problem
- Search problem
- Counting problem (complexity)
- Function problem

References

[1] Ausiello, G.; et al. (2003), *Complexity and Approximation* (Corrected edition ed.), Springer, ISBN 978-3540654315

[2] Hromkovic, Juraj (2002), *Algorithms for Hard Problems*, Texts in Theoretical Computer Science (2nd ed.), Springer, ISBN 978-3540441342

[3] Kann, Viggo (1992), *On the Approximability of NP-complete Optimization Problems*, Royal Institute of Technology, Sweden, ISBN 91-7170-082-X

Constraint satisfaction problem

Constraint satisfaction problems or **CSP**s are mathematical problems defined as a set of objects whose state must satisfy a number of *constraints* or limitations. CSPs represent the entities in a problem as a homogeneous collection of finite constraints over variables, which is solved by constraint satisfaction methods. CSPs are the subject of intense research in both artificial intelligence and operations research, since the regularity in their formulation provides a common basis to analyze and solve problems of many unrelated families. CSPs often exhibit high complexity, requiring a combination of heuristics and combinatorial search methods to be solved in a reasonable time.

Examples of problems that can be modeled as a constraint satisfaction problem:

- Eight queens puzzle
- Map coloring problem
- Sudoku
- Boolean satisfiability

Formal definition

Formally, a constraint satisfaction problem is defined as a triple $\langle X, D, C \rangle$, where X is a set of variables, D is a domain of values, and C is a set of constraints. Every constraint is in turn a pair $\langle t, R \rangle$, where t is a tuple of variables and R is a set of tuples of values; all these tuples having the same number of elements; as a result R is a relation. An evaluation of the variables is a function from variables to values, $v : X \to D$. Such an evaluation satisfies a constraint $\langle (x_1, \ldots, x_n), R \rangle$ if $(v(x_1), \ldots, v(x_n)) \in R$. A solution is an evaluation that satisfies all constraints.

Resolution of CSPs

Constraint satisfaction problems on finite domains are typically solved using a form of search. The most used techniques are variants of backtracking, constraint propagation, and local search.

Backtracking is a recursive algorithm. It maintains a partial assignment of the variables. Initially, all variables are unassigned. At each step, a variable is chosen, and all possible values are assigned to it in turn. For each value, the consistency of the partial assignment with the constraints is checked; in case of consistency, a recursive call is performed. When all values have been tried, the algorithm backtracks. In this basic backtracking algorithm, consistency is defined as the satisfaction of all constraints whose variables are all assigned. Several variants of backtracking exists. Backmarking improves the efficiency of checking consistency. Backjumping allows saving part of the search by backtracking "more than one variable" in some cases. Constraint learning infers and saves new constraints that can be later used to avoid part of the search. Look-ahead is also often used in backtracking to attempt to foresee the effects of choosing a variable or a value, thus sometimes determining in advance when a subproblem is satisfiable or unsatisfiable.

Constraint propagation techniques are methods used to modify a constraint satisfaction problem. More precisely, they are methods that enforce a form of local consistency, which are conditions related to the consistency of a group of variables and/or constraints. Constraints propagation has various uses. First, they turn a problem into one that is equivalent but is usually simpler to solve. Second, they may prove satisfiability or unsatisfiability of problems. This is not guaranteed to happen in general; however, it always happens for some forms of constraint propagation and/or for some certain kinds of problems. The most known and used form of local consistency are arc consistency, hyper-arc consistency, and path consistency. The most popular constraint propagation method is the AC-3 algorithm, which enforces arc consistency.

Local search methods are incomplete satisfiability algorithms. They may find a solution of a problem, but they may fail even if the problem is satisfiable. They work by iteratively improving a complete assignment over the variables. At each step, a small number of variables are changed value, with the overall aim of increasing the number of constraints satisfied by this assignment. The min-conflicts algorithm is a local search algorithm specific for CSPs and based in that principle. In practice, local search appears to work well when these changes are also affected by random choices. Integration of search with local search have been developed, leading to hybrid algorithms.

Theoretical aspects of CSPs

CSPs are also studied in computational complexity theory and finite model theory. An important question is whether for each set of relations, the set of all CSPs that can be represented using only relations chosen from that set is either in PTIME or otherwise NP-complete (assuming P ≠ NP). If such a dichotomy is true, then CSPs provide one of the largest known subsets of NP which avoids problems that are neither polynomial time solvable nor NP-complete, whose existence was demonstrated by Ladner. Dichotomy results are known for CSPs where the domain of values is of size 2 or 3, but the general case is still open.

Most classes of CSPs that are known to be tractable are those where the hypergraph of constraints has bounded treewidth (and there are no restrictions on the set of constraint relations), or where the constraints have arbitrary form but there exist essentially non-unary polymorphisms of the set of constraint relations.

Every CSP can also be considered as a conjunctive query containment problem[1] .

Variants of CSPs

The classic model of Constraint Satisfaction Problem defines a model of static, inflexible constraints. This rigid model is a shortcoming that makes it difficult to represent problems easily [2] . Several modifications of the basic CSP definition have been proposed to adapt the model to a wide variety of problems.

Dynamic CSPs

Dynamic CSPs[3] (*DCSP*s) are useful when the original formulation of a problem is altered in some way, typically because the set of constraints to consider evolves because of the environment.[4] DCSPs are viewed as a sequence of static CSPs, each one a transformation of the previous one in which variables and constraints can be added (restriction) or removed (relaxation). Information found in the initial formulations of the problem can be used to refine the next ones. The solving method can be classified according to the way in which information is transferred:

- Oracles: the solution found to previous CSPs in the sequence are used as heuristics to guide the resolution of the current CSP from scratch.
- Local repair: each CSP is calculated starting from the partial solution of the previous one and repairing the inconsistent constraints with local search.
- Constraint recording: new constraints are defined in each stage of the search to represent the learning of inconsistent group of decisions. Those constraints are carried over the new CSP problems.

Flexible CSPs

Classic CSPs treat constraints as hard, meaning that they are *imperative* (each solution must satisfy all them) and *inflexible* (in the sense that they must be completely satisfied or else they are completely violated). **Flexible CSP**s relax those assumptions, partially *relaxing* the constraints and allowing the solution to not comply with all them. Some types of flexible CSPs include:

- MAX-CSP, where a number of constraints are allowed to be violated, and the quality of a solution is measured by the number of satisfied constraints.
- Weighted CSP, a MAX-CSP in which each violation of a constraint is weighted according to a predefined preference. Thus satisfying constraint with more weight is preferred.
- Fuzzy CSP model constraints as fuzzy relations in which the satisfaction of a constraint is a continuous function of its variables' values, going from fully satisfied to fully violated.

Further reading

- Steven Minton, Andy Philips, Mark D. Johnston, Philip Laird (1993). "Minimizing Conflicts: A Heuristic Repair Method for Constraint-Satisfaction and Scheduling Problems" [5] (PDF). *Journal of Artificial Intelligence Research* **58**: 161–205.

See also

- Constraint satisfaction
- Declarative programming
- Constraint programming
- Distributed Constraint Satisfaction Problem (DisCSP)

External links

- CSP Tutorial [6]
- Tsang, Edward (1993). *Foundations of Constraint Satisfaction* [7]. Academic Press. ISBN 0-12-701610-4
- Dechter, Rina (2003). *Constraint processing* [8]. Morgan Kaufmann. ISBN 1-55860-890-7
- Apt, Krzysztof (2003). *Principles of constraint programming*. Cambridge University Press. ISBN 0-521-82583-0
- Lecoutre, Christophe (2009). *Constraint Networks: Techniques and Algorithms* [9]. ISTE/Wiley. ISBN 978-1-84821-106-3
- Tomás Feder, *Constraint satisfaction: a personal perspective* [10], manuscript.
- Constraints archive [11]
- Forced Satisfiable CSP Benchmarks of Model RB [12]
- Benchmarks -- XML representation of CSP instances [13]
- Dynamic Flexible Constraint Satisfaction and Its Application to AI Planning [14], Ian Miguel - slides.
- Constraint Propagation [15] - Dissertation by Guido Tack giving a good survey of theory and implementation issues

References

[1] Kolaitis, Phokion G.; Vardi, Moshe Y. (2000). "Conjunctive-Query Containment and Constraint Satisfaction". *Journal of Computer and System Sciences* **61**: 302–332. doi:10.1006/jcss.2000.1713.

[2] Dynamic Flexible Constraint Satisfaction and Its Application to AI Planning (http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.9.6733), Ian Miguel

[3] Dechter, R. and Dechter, A. Belief. Maintenance in Dynamic Constraint Networks In Proc. of AAAI-88, 37-42. (http://www.ics.uci.edu/~csp/r5.pdf)

[4] Solution reuse in dynamic constraint satisfaction problems (http://www.aaai.org/Papers/AAAI/1994/AAAI94-302.pdf), Thomas Schiex

[5] https://eprints.kfupm.edu.sa/50799/1/50799.pdf

[6] http://4c.ucc.ie/web/outreach/tutorial.html

[7] http://www.bracil.net/edward/FCS.html

[8] http://www.ics.uci.edu/~dechter/books/index.html

[9] http://www.iste.co.uk/index.php?f=a&ACTION=View&id=250

[10] http://theory.stanford.edu/~tomas/consmod.pdf

[11] http://4c.ucc.ie/web/archive/index.jsp

[12] http://www.nlsde.buaa.edu.cn/~kexu/benchmarks/benchmarks.htm

[13] http://www.cril.univ-artois.fr/~lecoutre/research/benchmarks/benchmarks.html

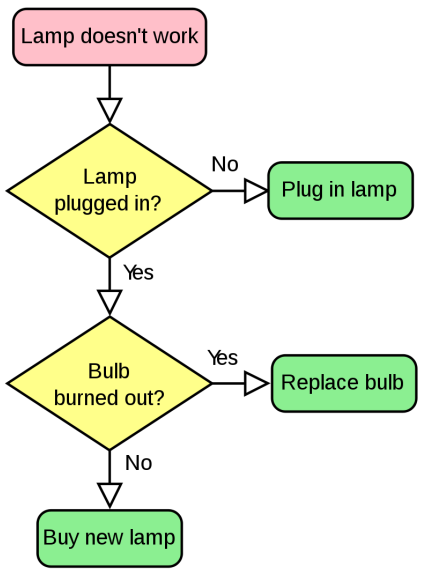
[14] http://www.cs.st-andrews.ac.uk/~ianm/docs/Thesis.ppt

[15] http://www.ps.uni-sb.de/Papers/abstracts/tackDiss.html

Algorithms

Algorithm

In mathematics, computer science, and related subjects, an **algorithm** is an effective method for solving a problem expressed as a finite sequence of instructions. Algorithms are used for calculation, data processing, and many other fields. (In more advanced or abstract settings, the instructions do not necessarily constitute a finite sequence, and even not necessarily a sequence; see, e.g., "nondeterministic algorithm".)



This is an algorithm that tries to figure out why the lamp doesn't turn on and tries to fix it using the steps. Flowcharts are often used to graphically represent algorithms.

Each algorithm is a list of well-defined instructions for completing a task. Starting from an initial state, the instructions describe a computation that proceeds through a well-defined series of successive states, eventually terminating in a final ending state. The transition from one state to the next is not necessarily deterministic; some algorithms, known as randomized algorithms, incorporate randomness.

A partial formalization of the concept began with attempts to solve the Entscheidungsproblem (the "decision problem") posed by David Hilbert in 1928. Subsequent formalizations were framed as attempts to define "effective calculability"[1] or "effective method";[2] those formalizations included the Gödel–Herbrand–Kleene recursive functions of 1930, 1934 and 1935, Alonzo Church's lambda calculus of 1936, Emil Post's "Formulation 1" of 1936, and Alan Turing's Turing machines of 1936–7 and 1939.

The adjective "continuous" when applied to the word "algorithm" can mean: 1) An algorithm operating on data that represents continuous quantities, even though this data is represented by discrete approximations – such algorithms are studied in numerical analysis; or 2) An algorithm in the form of a differential equation that operates continuously on the data, running on an analog computer.[3]

Etymology

Al-Khwārizmī, Muslim Persian astronomer and mathematician, wrote a treatise in the Arabic language in 825 AD, *On Calculation with Hindu–Arabic numeral system*. (See algorism). It was translated from Arabic into Latin in the 12th century as *Algoritmi de numero Indorum* (al-Daffa 1977), whose title is supposedly likely intended to mean "Algoritmi on the numbers of the Indians", where "Algoritmi" was the translator's rendition of the author's name; but people misunderstanding the title treated *Algoritmi* as a Latin plural and this led to the word "algorithm" (Latin *algorismus*) coming to mean "calculation method". The intrusive "th" is most likely due to a false cognate with the Greek ἀριθμός (*arithmos*) meaning "numbers".

Why algorithms are necessary: an informal definition

For a detailed presentation of the various points of view around the definition of "algorithm" see Algorithm characterizations. For examples of simple addition algorithms specified in the detailed manner described in Algorithm characterizations, see Algorithm examples.

While there is no generally accepted *formal* definition of "algorithm," an informal definition could be "a process that performs some sequence of operations." For some people, a program is only an algorithm if it stops eventually. For others, a program is only an algorithm if it stops before a given number of calculation steps.

A prototypical example of an algorithm is Euclid's algorithm to determine the maximum common divisor of two integers.

We can derive clues to the issues involved and an informal meaning of the word from the following quotation from Boolos & Jeffrey (1974, 1999) (boldface added):

> No human being can write fast enough, or long enough, or small enough† (†"smaller and smaller without limit ...you'd be trying to write on molecules, on atoms, on electrons") to list all members of an enumerably infinite set by writing out their names, one after another, in some notation. But humans can do something equally useful, in the case of certain enumerably infinite sets: They can give **explicit instructions for determining the *n*th member of the set**, for arbitrary finite *n*. Such instructions are to be given quite explicitly, in a form in which **they could be followed by a computing machine**, or by a **human who is capable of carrying out only very elementary operations on symbols**[4]

The term "enumerably infinite" means "countable using integers perhaps extending to infinity." Thus Boolos and Jeffrey are saying that an algorithm *implies* instructions for a process that "creates" output integers from an *arbitrary* "input" integer or integers that, in theory, can be chosen from 0 to infinity. Thus we might expect an algorithm to be an algebraic equation such as **y = m + n** — two arbitrary "input variables" **m** and **n** that produce an output **y**. As we see in Algorithm characterizations — the word algorithm implies much more than this, something on the order of (for our addition example):

> Precise instructions (in language understood by "the computer") for a "fast, efficient, good" *process* that specifies the "moves" of "the computer" (machine or human, equipped with the necessary internally-contained information and capabilities) to find, decode, and then munch arbitrary input integers/symbols **m** and **n**, symbols **+** and **=** ... and (reliably, correctly, "effectively") produce, in a "reasonable" time, output-integer **y** at a specified place and in a specified format.

The concept of *algorithm* is also used to define the notion of decidability. That notion is central for explaining how formal systems come into being starting from a small set of axioms and rules. In logic, the time that an algorithm requires to complete cannot be measured, as it is not apparently related with our customary physical dimension. From such uncertainties, that characterize ongoing work, stems the unavailability of a definition of *algorithm* that suits both concrete (in some sense) and abstract usage of the term.

Formalization

Algorithms are essential to the way computers process information. Many computer programs contain algorithms that specify the specific instructions a computer should perform (in a specific order) to carry out a specified task, such as calculating employees' paychecks or printing students' report cards. Thus, an algorithm can be considered to be any sequence of operations that can be simulated by a Turing-complete system. Authors who assert this thesis include Minsky (1967), Savage (1987) and Gurevich (2000):

> Minsky: "But we will also maintain, with Turing . . . that any procedure which could "naturally" be called effective, can in fact be realized by a (simple) machine. Although this may seem extreme, the arguments . . . in its favor are hard to refute".[5]

> Gurevich: "...Turing's informal argument in favor of his thesis justifies a stronger thesis: every algorithm can be simulated by a Turing machine ... according to Savage [1987], an algorithm is a computational process defined by a Turing machine".[6]

Typically, when an algorithm is associated with processing information, data is read from an input source, written to an output device, and/or stored for further processing. Stored data is regarded as part of the internal state of the entity performing the algorithm. In practice, the state is stored in one or more data structures.

For any such computational process, the algorithm must be rigorously defined: specified in the way it applies in all possible circumstances that could arise. That is, any conditional steps must be systematically dealt with, case-by-case; the criteria for each case must be clear (and computable).

Because an algorithm is a precise list of precise steps, the order of computation will always be critical to the functioning of the algorithm. Instructions are usually assumed to be listed explicitly, and are described as starting "from the top" and going "down to the bottom", an idea that is described more formally by *flow of control*.

So far, this discussion of the formalization of an algorithm has assumed the premises of imperative programming. This is the most common conception, and it attempts to describe a task in discrete, "mechanical" means. Unique to this conception of formalized algorithms is the assignment operation, setting the value of a variable. It derives from the intuition of "memory" as a scratchpad. There is an example below of such an assignment.

For some alternate conceptions of what constitutes an algorithm see functional programming and logic programming .

Termination

Some writers restrict the definition of *algorithm* to procedures that eventually finish. In such a category Kleene places the "*decision procedure* or *decision method* or *algorithm* for the question".[7] Others, including Kleene, include procedures that could run forever without stopping; such a procedure has been called a "computational method"[8] or "*calculation procedure* or *algorithm* (and hence a *calculation problem*) in relation to a general question which requires for an answer, not yes or no, but **the exhibiting of some object**".[9]

Minsky makes the pertinent observation, in regards to determining whether an algorithm will eventually terminate (from a particular starting state):

> But if the length of the process isn't known in advance, then "trying" it may not be decisive, because if the process does go on forever — then at no time will we ever be sure of the answer.[5]

As it happens, no other method can do any better, as was shown by Alan Turing with his celebrated result on the undecidability of the so-called halting problem. There is no algorithmic procedure for determining of arbitrary algorithms whether or not they terminate from given starting states. The analysis of algorithms for their likelihood of termination is called termination analysis.

See the examples of (im-)"proper" subtraction at partial function for more about what can happen when an algorithm fails for certain of its input numbers — e.g., (i) non-termination, (ii) production of "junk" (output in the wrong format to be considered a number) or no number(s) at all (halt ends the computation with no output), (iii) wrong number(s), or (iv) a combination of these. Kleene proposed that the production of "junk" or failure to produce a number is solved by having the algorithm detect these instances and produce e.g., an error message (he suggested "0"), or preferably, force the algorithm into an endless loop.[10] Davis (1958) does this to his subtraction algorithm — he fixes his algorithm in a second example so that it is proper subtraction and it terminates.[11] Along with the logical outcomes "true" and "false" Kleene (1952) also proposes the use of a third logical symbol "u" — undecided[12] — thus an algorithm will always produce *something* when confronted with a "proposition". The problem of wrong answers must be solved with an independent "proof" of the algorithm e.g., using induction:

> We normally require auxiliary evidence for this [that the algorithm correctly defines a mu recursive function], e.g., in the form of an inductive proof that, for each argument value, the computation

terminates with a unique value.[13]

Expressing algorithms

Algorithms can be expressed in many kinds of notation, including natural languages, pseudocode, flowcharts, programming languages or control tables (processed by interpreters). Natural language expressions of algorithms tend to be verbose and ambiguous, and are rarely used for complex or technical algorithms. Pseudocode, flowcharts and control tables are structured ways to express algorithms that avoid many of the ambiguities common in natural language statements, while remaining independent of a particular implementation language. Programming languages are primarily intended for expressing algorithms in a form that can be executed by a computer, but are often used as a way to define or document algorithms.

There is a wide variety of representations possible and one can express a given Turing machine program as a sequence of machine tables (see more at finite state machine and state transition table), as flowcharts (see more at state diagram), or as a form of rudimentary machine code or assembly code called "sets of quadruples" (see more at Turing machine).

Sometimes it is helpful in the description of an algorithm to supplement small "flow charts" (state diagrams) with natural-language and/or arithmetic expressions written inside "block diagrams" to summarize what the "flow charts" are accomplishing.

Representations of algorithms are generally classed into three accepted levels of Turing machine description:[14]

- **1 High-level description**:

  "...prose to describe an algorithm, ignoring the implementation details. At this level we do not need to mention how the machine manages its tape or head."

- **2 Implementation description**:

  "...prose used to define the way the Turing machine uses its head and the way that it stores data on its tape. At this level we do not give details of states or transition function."

- **3 Formal description**:

  Most detailed, "lowest level", gives the Turing machine's "state table".

For an example of the simple algorithm "Add m+n" described in all three levels see Algorithm examples.

Computer algorithms

In computer systems, an algorithm is basically an instance of logic written in software by software developers to be effective for the intended "target" computer(s), in order for the software on the target machines to *do something*. For instance, if a person is writing software that is supposed to print out a PDF document located at the operating system folder "/My Documents" at computer drive "D:" every Friday at 10PM, they will write an algorithm that specifies the following actions: "If today's date (computer time) is 'Friday,' open the document at 'D:/My Documents' and call the 'print' function". While this simple algorithm does not look into whether the printer has enough paper or whether the document has been moved into a different location, one can make this algorithm more robust and anticipate these problems by rewriting it as a formal CASE statement[15] or as a (carefully crafted) sequence of IF-THEN-ELSE statements.[16] For example the CASE statement might appear as follows (there are other possibilities):

CASE 1: IF today's date is NOT Friday THEN *exit this CASE instruction* ELSE

CASE 2: IF today's date is Friday AND the document is located at 'D:/My Documents' AND there is paper in the printer THEN print the document (and *exit this CASE instruction*) ELSE

CASE 3: IF today's date is Friday AND the document is NOT located at 'D:/My Documents' THEN display 'document not found' error message (and *exit this CASE instruction*) ELSE

CASE 4: IF today's date is Friday AND the document is located at 'D:/My Documents' AND there is NO paper in the printer THEN (i) display 'out of paper' error message and (ii) *exit*.

Note that CASE 3 includes two possibilities: (i) the document is NOT located at 'D:/My Documents' AND there's paper in the printer OR (ii) the document is NOT located at 'D:/My Documents' AND there's NO paper in the printer.

The sequence of IF-THEN-ELSE tests might look like this:

TEST 1: IF today's date is NOT Friday THEN *done* ELSE TEST 2:

TEST 2: IF the document is NOT located at 'D:/My Documents' THEN display 'document not found' error message ELSE TEST 3:

TEST 3: IF there is NO paper in the printer THEN display 'out of paper' error message ELSE print the document.

These examples' logic grants precedence to the instance of "NO document at 'D:/My Documents' ". Also observe that in a well-crafted CASE statement or sequence of IF-THEN-ELSE statements the number of distinct actions—4 in these examples: do nothing, print the document, display 'document not found', display 'out of paper' – equals the number of cases.

Given unlimited memory, a computational machine with the ability to execute either a set of CASE statements or a sequence of IF-THEN-ELSE statements is Turing complete. Therefore, anything that is computable can be computed by this machine. This form of algorithm is fundamental to computer programming in all its forms (see more at McCarthy formalism).

Implementation

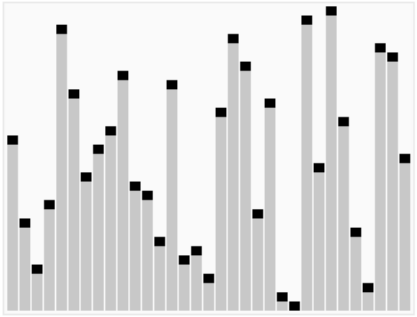
Most algorithms are intended to be implemented as computer programs. However, algorithms are also implemented by other means, such as in a biological neural network (for example, the human brain implementing arithmetic or an insect looking for food), in an electrical circuit, or in a mechanical device.

Example

One of the simplest algorithms is to find the largest number in an (unsorted) list of numbers. The solution necessarily requires looking at every number in the list, but only once at each. From this follows a simple algorithm, which can be stated in a high-level description English prose, as:

High-level description:

1. Assume the first item is largest.
2. Look at each of the remaining items in the list and if it is larger than the largest item so far, make a note of it.
3. The last noted item is the largest in the list when the process is complete.



An animation of the quicksort algorithm sorting an array of randomized values. The red bars mark the pivot element; at the start of the animation, the element farthest to the right hand side is chosen as the pivot.

(Quasi-)formal description: Written in prose but much closer to the high-level language of a computer program, the following is the more formal coding of the algorithm in pseudocode or pidgin code:

```
Algorithm LargestNumber
  Input: A non-empty list of numbers L.

  Output: The largest number in the list L.
```

```
largest ← L0
for each item in the list (Length(L)≥1), do
  if the item > largest, then
    largest ← the item
return largest
```

- "←" is a loose shorthand for "changes to". For instance, "*largest* ← *item*" means that the value of *largest* changes to the value of *item*.
- "**return**" terminates the algorithm and outputs the value that follows.

For a more complex example of an algorithm, see Euclid's algorithm for the greatest common divisor, one of the earliest algorithms known.

Algorithmic analysis

It is frequently important to know how much of a particular resource (such as time or storage) is theoretically required for a given algorithm. Methods have been developed for the analysis of algorithms to obtain such quantitative answers (estimates); for example, the algorithm above has a time requirement of O(*n*), using the big O notation with *n* as the length of the list. At all times the algorithm only needs to remember two values: the largest number found so far, and its current position in the input list. Therefore it is said to have a space requirement of *O(1)*, if the space required to store the input numbers is not counted, or O(*n*) if it is counted.

Different algorithms may complete the same task with a different set of instructions in less or more time, space, or 'effort' than others. For example, a binary search algorithm will usually outperform a brute force sequential search when used for table lookups on sorted lists.

Formal versus empirical

The analysis and study of algorithms is a discipline of computer science, and is often practiced abstractly without the use of a specific programming language or implementation. In this sense, algorithm analysis resembles other mathematical disciplines in that it focuses on the underlying properties of the algorithm and not on the specifics of any particular implementation. Usually pseudocode is used for analysis as it is the simplest and most general representation. However, ultimately, most algorithms are usually implemented on particular hardware / software platforms and their algorithmic efficiency is eventually put to the test using real code.

Empirical testing is useful because it may uncover unexpected interactions that affect performance. For instance an algorithm that has no locality of reference may have much poorer performance than predicted because it 'thrashes the cache'. Benchmarks may be used to compare before/after potential improvements to an algorithm after program optimization.

Classification

There are various ways to classify algorithms, each with its own merits.

By implementation

One way to classify algorithms is by implementation means.

- **Recursion** or **iteration**: A recursive algorithm is one that invokes (makes reference to) itself repeatedly until a certain condition matches, which is a method common to functional programming. Iterative algorithms use repetitive constructs like loops and sometimes additional data structures like stacks to solve the given problems. Some problems are naturally suited for one implementation or the other. For example, towers of Hanoi is well understood in recursive implementation. Every recursive version has an equivalent (but possibly more or less complex) iterative version, and vice versa.

- **Logical**: An algorithm may be viewed as controlled logical deduction. This notion may be expressed as: **Algorithm = logic + control**.[17] The logic component expresses the axioms that may be used in the computation and the control component determines the way in which deduction is applied to the axioms. This is the basis for the logic programming paradigm. In pure logic programming languages the control component is fixed and algorithms are specified by supplying only the logic component. The appeal of this approach is the elegant semantics: a change in the axioms has a well defined change in the algorithm.
- **Serial** or **parallel** or **distributed**: Algorithms are usually discussed with the assumption that computers execute one instruction of an algorithm at a time. Those computers are sometimes called serial computers. An algorithm designed for such an environment is called a serial algorithm, as opposed to parallel algorithms or distributed algorithms. Parallel algorithms take advantage of computer architectures where several processors can work on a problem at the same time, whereas distributed algorithms utilize multiple machines connected with a network. Parallel or distributed algorithms divide the problem into more symmetrical or asymmetrical subproblems and collect the results back together. The resource consumption in such algorithms is not only processor cycles on each processor but also the communication overhead between the processors. Sorting algorithms can be parallelized efficiently, but their communication overhead is expensive. Iterative algorithms are generally parallelizable. Some problems have no parallel algorithms, and are called inherently serial problems.
- **Deterministic** or **non-deterministic**: Deterministic algorithms solve the problem with exact decision at every step of the algorithm whereas non-deterministic algorithms solve problems via guessing although typical guesses are made more accurate through the use of heuristics.
- **Exact** or **approximate**: While many algorithms reach an exact solution, approximation algorithms seek an approximation that is close to the true solution. Approximation may use either a deterministic or a random strategy. Such algorithms have practical value for many hard problems.

By design paradigm

Another way of classifying algorithms is by their design methodology or paradigm. There is a certain number of paradigms, each different from the other. Furthermore, each of these categories will include many different types of algorithms. Some commonly found paradigms include:

- **Brute-force** or **exhaustive search**. This is the naïve method of trying every possible solution to see which is best.[18]
- **Divide and conquer**. A divide and conquer algorithm repeatedly reduces an instance of a problem to one or more smaller instances of the same problem (usually recursively) until the instances are small enough to solve easily. One such example of divide and conquer is merge sorting. Sorting can be done on each segment of data after dividing data into segments and sorting of entire data can be obtained in the conquer phase by merging the segments. A simpler variant of divide and conquer is called a **decrease and conquer algorithm**, that solves an identical subproblem and uses the solution of this subproblem to solve the bigger problem. Divide and conquer divides the problem into multiple subproblems and so the conquer stage will be more complex than decrease and conquer algorithms. An example of decrease and conquer algorithm is the binary search algorithm.
- **Dynamic programming**. When a problem shows optimal substructure, meaning the optimal solution to a problem can be constructed from optimal solutions to subproblems, and overlapping subproblems, meaning the same subproblems are used to solve many different problem instances, a quicker approach called *dynamic programming* avoids recomputing solutions that have already been computed. For example, the shortest path to a goal from a vertex in a weighted graph can be found by using the shortest path to the goal from all adjacent vertices. Dynamic programming and memoization go together. The main difference between dynamic programming and divide and conquer is that subproblems are more or less independent in divide and conquer, whereas subproblems overlap in dynamic programming. The difference between dynamic programming and straightforward recursion is in caching or memoization of recursive calls. When subproblems are independent and there is no repetition, memoization does not help; hence dynamic programming is not a solution for all complex

problems. By using memoization or maintaining a table of subproblems already solved, dynamic programming reduces the exponential nature of many problems to polynomial complexity.

- **The greedy method**. A greedy algorithm is similar to a dynamic programming algorithm, but the difference is that solutions to the subproblems do not have to be known at each stage; instead a "greedy" choice can be made of what looks best for the moment. The greedy method extends the solution with the best possible decision (not all feasible decisions) at an algorithmic stage based on the current local optimum and the best decision (not all possible decisions) made in a previous stage. It is not exhaustive, and does not give accurate answer to many problems. But when it works, it will be the fastest method. The most popular greedy algorithm is finding the minimal spanning tree as given by Kruskal.
- **Linear programming**. When solving a problem using linear programming, specific inequalities involving the inputs are found and then an attempt is made to maximize (or minimize) some linear function of the inputs. Many problems (such as the maximum flow for directed graphs) can be stated in a linear programming way, and then be solved by a 'generic' algorithm such as the simplex algorithm. A more complex variant of linear programming is called integer programming, where the solution space is restricted to the integers.
- **Reduction**. This technique involves solving a difficult problem by transforming it into a better known problem for which we have (hopefully) asymptotically optimal algorithms. The goal is to find a reducing algorithm whose complexity is not dominated by the resulting reduced algorithm's. For example, one selection algorithm for finding the median in an unsorted list involves first sorting the list (the expensive portion) and then pulling out the middle element in the sorted list (the cheap portion). This technique is also known as *transform and conquer*.
- **Search and enumeration**. Many problems (such as playing chess) can be modeled as problems on graphs. A graph exploration algorithm specifies rules for moving around a graph and is useful for such problems. This category also includes search algorithms, branch and bound enumeration and backtracking.

1. Randomized algorithms are those that make some choices randomly (or pseudo-randomly); for some problems, it can in fact be proven that the fastest solutions must involve some randomness. There are two large classes of such algorithms:
    1. Monte Carlo algorithms return a correct answer with high-probability. E.g. RP is the subclass of these that run in polynomial time)
    2. Las Vegas algorithms always return the correct answer, but their running time is only probabilistically bound, e.g. ZPP.
2. In optimization problems, heuristic algorithms do not try to find an optimal solution, but an approximate solution where the time or resources are limited. They are not practical to find perfect solutions. An example of this would be local search, tabu search, or simulated annealing algorithms, a class of heuristic probabilistic algorithms that vary the solution of a problem by a random amount. The name "simulated annealing" alludes to the metallurgic term meaning the heating and cooling of metal to achieve freedom from defects. The purpose of the random variance is to find close to globally optimal solutions rather than simply locally optimal ones, the idea being that the random element will be decreased as the algorithm settles down to a solution. Approximation algorithms are those heuristic algorithms that additionally provide some bounds on the error. Genetic algorithms attempt to find solutions to problems by mimicking biological evolutionary processes, with a cycle of random mutations yielding successive generations of "solutions". Thus, they emulate reproduction and "survival of the fittest". In genetic programming, this approach is extended to algorithms, by regarding the algorithm itself as a "solution" to a problem.

By field of study

Every field of science has its own problems and needs efficient algorithms. Related problems in one field are often studied together. Some example classes are search algorithms, sorting algorithms, merge algorithms, numerical algorithms, graph algorithms, string algorithms, computational geometric algorithms, combinatorial algorithms, machine learning, cryptography, data compression algorithms and parsing techniques.

Fields tend to overlap with each other, and algorithm advances in one field may improve those of other, sometimes completely unrelated, fields. For example, dynamic programming was invented for optimization of resource consumption in industry, but is now used in solving a broad range of problems in many fields.

By complexity

Algorithms can be classified by the amount of time they need to complete compared to their input size. There is a wide variety: some algorithms complete in linear time relative to input size, some do so in an exponential amount of time or even worse, and some never halt. Additionally, some problems may have multiple algorithms of differing complexity, while other problems might have no algorithms or no known efficient algorithms. There are also mappings from some problems to other problems. Owing to this, it was found to be more suitable to classify the problems themselves instead of the algorithms into equivalence classes based on the complexity of the best possible algorithms for them.

By computing power

Another way to classify algorithms is by computing power. This is typically done by considering some collection (class) of algorithms. A recursive class of algorithms is one that includes algorithms for all Turing computable functions. Looking at classes of algorithms allows for the possibility of restricting the available computational resources (time and memory) used in a computation. A subrecursive class of algorithms is one in which not all Turing computable functions can be obtained. For example, the algorithms that run in polynomial time suffice for many important types of computation but do not exhaust all Turing computable functions. The class of algorithms implemented by primitive recursive functions is another subrecursive class.

Burgin (2005, p. 24) uses a generalized definition of algorithms that relaxes the common requirement that the output of the algorithm that computes a function must be determined after a finite number of steps. He defines a super-recursive class of algorithms as "a class of algorithms in which it is possible to compute functions not computable by any Turing machine" (Burgin 2005, p. 107). This is closely related to the study of methods of hypercomputation.

Legal issues

> *See also: Software patents for a general overview of the patentability of software, including computer-implemented algorithms.*

Algorithms, by themselves, are not usually patentable. In the United States, a claim consisting solely of simple manipulations of abstract concepts, numbers, or signals does not constitute "processes" (USPTO 2006), and hence algorithms are not patentable (as in Gottschalk v. Benson). However, practical applications of algorithms are sometimes patentable. For example, in Diamond v. Diehr, the application of a simple feedback algorithm to aid in the curing of synthetic rubber was deemed patentable. The patenting of software is highly controversial, and there are highly criticized patents involving algorithms, especially data compression algorithms, such as Unisys' LZW patent.

Additionally, some cryptographic algorithms have export restrictions (see export of cryptography).

History: Development of the notion of "algorithm"

Discrete and distinguishable symbols

Tally-marks: To keep track of their flocks, their sacks of grain and their money the ancients used tallying: accumulating stones or marks scratched on sticks, or making discrete symbols in clay. Through the Babylonian and Egyptian use of marks and symbols, eventually Roman numerals and the abacus evolved (Dilson, p. 16–41). Tally marks appear prominently in unary numeral system arithmetic used in Turing machine and Post–Turing machine computations.

Manipulation of symbols as "place holders" for numbers: algebra

The work of the ancient Greek geometers, Persian mathematician Al-Khwarizmi (often considered the "father of algebra" and from whose name the terms "algorism" and "algorithm" are derived), and Western European mathematicians culminated in Leibniz's notion of the calculus ratiocinator (ca 1680):

> A good century and a half ahead of his time, Leibniz proposed an algebra of logic, an algebra that would specify the rules for manipulating logical concepts in the manner that ordinary algebra specifies the rules for manipulating numbers.[19]

Mechanical contrivances with discrete states

The clock: Bolter credits the invention of the weight-driven clock as "The key invention [of Europe in the Middle Ages]", in particular the verge escapement[20] that provides us with the tick and tock of a mechanical clock. "The accurate automatic machine"[21] led immediately to "mechanical automata" beginning in the thirteenth century and finally to "computational machines" – the difference engine and analytical engines of Charles Babbage and Countess Ada Lovelace.[22]

Logical machines 1870 – Stanley Jevons' "logical abacus" and "logical machine": The technical problem was to reduce Boolean equations when presented in a form similar to what are now known as Karnaugh maps. Jevons (1880) describes first a simple "abacus" of "slips of wood furnished with pins, contrived so that any part or class of the [logical] combinations can be picked out mechanically . . . More recently however I have reduced the system to a completely mechanical form, and have thus embodied the whole of the indirect process of inference in what may be called a **Logical Machine**" His machine came equipped with "certain moveable wooden rods" and "at the foot are 21 keys like those of a piano [etc] . . .". With this machine he could analyze a "syllogism or any other simple logical argument".[23]

This machine he displayed in 1870 before the Fellows of the Royal Society.[24] Another logician John Venn, however, in his 1881 *Symbolic Logic*, turned a jaundiced eye to this effort: "I have no high estimate myself of the interest or importance of what are sometimes called logical machines ... it does not seem to me that any contrivances at present known or likely to be discovered really deserve the name of logical machines"; see more at Algorithm characterizations. But not to be outdone he too presented "a plan somewhat analogous, I apprehend, to Prof. Jevon's *abacus* ... [And] [a]gain, corresponding to Prof. Jevons's logical machine, the following contrivance may be described. I prefer to call it merely a logical-diagram machine ... but I suppose that it could do very completely all that can be rationally expected of any logical machine".[25]

Jacquard loom, Hollerith punch cards, telegraphy and telephony — the electromechanical relay: Bell and Newell (1971) indicate that the Jacquard loom (1801), precursor to Hollerith cards (punch cards, 1887), and "telephone switching technologies" were the roots of a tree leading to the development of the first computers.[26] By the mid-1800s the telegraph, the precursor of the telephone, was in use throughout the world, its discrete and distinguishable encoding of letters as "dots and dashes" a common sound. By the late 1800s the ticker tape (ca 1870s) was in use, as was the use of Hollerith cards in the 1890 U.S. census. Then came the Teletype (ca. 1910) with its punched-paper use of Baudot code on tape.

Telephone-switching networks of electromechanical relays (invented 1835) was behind the work of George Stibitz (1937), the inventor of the digital adding device. As he worked in Bell Laboratories, he observed the "burdensome' use of mechanical calculators with gears. "He went home one evening in 1937 intending to test his idea... When the tinkering was over, Stibitz had constructed a binary adding device".[27]

Davis (2000) observes the particular importance of the electromechanical relay (with its two "binary states" *open* and *closed*):

> It was only with the development, beginning in the 1930s, of electromechanical calculators using electrical relays, that machines were built having the scope Babbage had envisioned."[28]

Mathematics during the 1800s up to the mid-1900s

Symbols and rules: In rapid succession the mathematics of George Boole (1847, 1854), Gottlob Frege (1879), and Giuseppe Peano (1888–1889) reduced arithmetic to a sequence of symbols manipulated by rules. Peano's *The principles of arithmetic, presented by a new method* (1888) was "the first attempt at an axiomatization of mathematics in a symbolic language".[29]

But Heijenoort gives Frege (1879) this kudos: Frege's is "perhaps the most important single work ever written in logic. ... in which we see a " 'formula language', that is a *lingua characterica*, a language written with special symbols, "for pure thought", that is, free from rhetorical embellishments ... constructed from specific symbols that are manipulated according to definite rules".[30] The work of Frege was further simplified and amplified by Alfred North Whitehead and Bertrand Russell in their Principia Mathematica (1910–1913).

The paradoxes: At the same time a number of disturbing paradoxes appeared in the literature, in particular the Burali-Forti paradox (1897), the Russell paradox (1902–03), and the Richard Paradox.[31] The resultant considerations led to Kurt Gödel's paper (1931) — he specifically cites the paradox of the liar — that completely reduces rules of recursion to numbers.

Effective calculability: In an effort to solve the Entscheidungsproblem defined precisely by Hilbert in 1928, mathematicians first set about to define what was meant by an "effective method" or "effective calculation" or "effective calculability" (i.e., a calculation that would succeed). In rapid succession the following appeared: Alonzo Church, Stephen Kleene and J.B. Rosser's λ-calculus[32] a finely-honed definition of "general recursion" from the work of Gödel acting on suggestions of Jacques Herbrand (cf. Gödel's Princeton lectures of 1934) and subsequent simplifications by Kleene.[33] Church's proof[34] that the Entscheidungsproblem was unsolvable, Emil Post's definition of effective calculability as a worker mindlessly following a list of instructions to move left or right through a sequence of rooms and while there either mark or erase a paper or observe the paper and make a yes-no decision about the next instruction.[35] Alan Turing's proof of that the Entscheidungsproblem was unsolvable by use of his "a- [automatic-] machine"[36] – in effect almost identical to Post's "formulation", J. Barkley Rosser's definition of "effective method" in terms of "a machine".[37] S. C. Kleene's proposal of a precursor to "Church thesis" that he called "Thesis I",[38] and a few years later Kleene's renaming his Thesis "Church's Thesis"[39] and proposing "Turing's Thesis".[40]

Emil Post (1936) and Alan Turing (1936–7, 1939)

Here is a remarkable coincidence of two men not knowing each other but describing a process of men-as-computers working on computations — and they yield virtually identical definitions.

Emil Post (1936) described the actions of a "computer" (human being) as follows:

> "...two concepts are involved: that of a *symbol space* in which the work leading from problem to answer is to be carried out, and a fixed unalterable *set of directions*.

His symbol space would be

"a two way infinite sequence of spaces or boxes... The problem solver or worker is to move and work in this symbol space, being capable of being in, and operating in but one box at a time.... a box is to admit of but two possible conditions, i.e., being empty or unmarked, and having a single mark in it, say a vertical stroke.

"One box is to be singled out and called the starting point. ...a specific problem is to be given in symbolic form by a finite number of boxes [i.e., INPUT] being marked with a stroke. Likewise the answer [i.e., OUTPUT] is to be given in symbolic form by such a configuration of marked boxes....

"A set of directions applicable to a general problem sets up a deterministic process when applied to each specific problem. This process will terminate only when it comes to the direction of type (C) [i.e., STOP]".[41]
See more at Post–Turing machine

Alan Turing's work[42] preceded that of Stibitz (1937); it is unknown whether Stibitz knew of the work of Turing. Turing's biographer believed that Turing's use of a typewriter-like model derived from a youthful interest: "Alan had dreamt of inventing typewriters as a boy; Mrs. Turing had a typewriter; and he could well have begun by asking himself what was meant by calling a typewriter 'mechanical'".[43] Given the prevalence of Morse code and telegraphy, ticker tape machines, and Teletypes we might conjecture that all were influences.

Turing — his model of computation is now called a Turing machine — begins, as did Post, with an analysis of a human computer that he whittles down to a simple set of basic motions and "states of mind". But he continues a step further and creates a machine as a model of computation of numbers.[44]

"Computing is normally done by writing certain symbols on paper. We may suppose this paper is divided into squares like a child's arithmetic book....I assume then that the computation is carried out on one-dimensional paper, i.e., on a tape divided into squares. I shall also suppose that the number of symbols which may be printed is finite....

"The behavior of the computer at any moment is determined by the symbols which he is observing, and his "state of mind" at that moment. We may suppose that there is a bound B to the number of symbols or squares which the computer can observe at one moment. If he wishes to observe more, he must use successive observations. We will also suppose that the number of states of mind which need be taken into account is finite...

"Let us imagine that the operations performed by the computer to be split up into 'simple operations' which are so elementary that it is not easy to imagine them further divided".[45]

Turing's reduction yields the following:

"The simple operations must therefore include:

"(a) Changes of the symbol on one of the observed squares

"(b) Changes of one of the squares observed to another square within L squares of one of the previously observed squares.

"It may be that some of these change necessarily invoke a change of state of mind. The most general single operation must therefore be taken to be one of the following:

"(A) A possible change (a) of symbol together with a possible change of state of mind.

"(B) A possible change (b) of observed squares, together with a possible change of state of mind"

"We may now construct a machine to do the work of this computer"[45] .

A few years later, Turing expanded his analysis (thesis, definition) with this forceful expression of it:

"A function is said to be "effectively calculable" if its values can be found by some purely mechanical process. Although it is fairly easy to get an intuitive grasp of this idea, it is nevertheless desirable to have some more definite, mathematical expressible definition . . . [he discusses the history of the definition pretty much as presented above with respect to Gödel, Herbrand, Kleene, Church, Turing and Post] . . . We may take this statement literally, understanding by a purely mechanical process one which could be carried out by a

machine. It is possible to give a mathematical description, in a certain normal form, of the structures of these machines. The development of these ideas leads to the author's definition of a computable function, and to an identification of computability † with effective calculability . . . .

"† We shall use the expression "computable function" to mean a function calculable by a machine, and we let "effectively calculable" refer to the intuitive idea without particular identification with any one of these definitions".[46]

J. B. Rosser (1939) and S. C. Kleene (1943)

J. Barkley Rosser boldly defined an 'effective [mathematical] method' in the following manner (boldface added):

"'Effective method' is used here in the rather special sense of a method each step of which is precisely determined and which is certain to produce the answer in a finite number of steps. With this special meaning, three different precise definitions have been given to date. [his footnote #5; see discussion immediately below]. The simplest of these to state (due to Post and Turing) says essentially that **an effective method of solving certain sets of problems exists if one can build a machine which will then solve any problem of the set with no human intervention beyond inserting the question and (later) reading the answer**. All three definitions are equivalent, so it doesn't matter which one is used. Moreover, the fact that all three are equivalent is a very strong argument for the correctness of any one." (Rosser 1939:225–6)

Rosser's footnote #5 references the work of (1) Church and Kleene and their definition of λ-definability, in particular Church's use of it in his *An Unsolvable Problem of Elementary Number Theory* (1936); (2) Herbrand and Gödel and their use of recursion in particular Gödel's use in his famous paper *On Formally Undecidable Propositions of Principia Mathematica and Related Systems I* (1931); and (3) Post (1936) and Turing (1936–7) in their mechanism-models of computation.

Stephen C. Kleene defined as his now-famous "Thesis I" known as the Church–Turing thesis. But he did this in the following context (boldface in original):

"12. **Algorithmic theories**... In setting up a complete algorithmic theory, what we do is to describe a procedure, performable for each set of values of the independent variables, which procedure necessarily terminates and in such manner that from the outcome we can read a definite answer, "yes" or "no," to the question, "is the predicate value true?"" (Kleene 1943:273)

History after 1950

A number of efforts have been directed toward further refinement of the definition of "algorithm", and activity is on-going because of issues surrounding, in particular, foundations of mathematics (especially the Church–Turing Thesis) and philosophy of mind (especially arguments around artificial intelligence). For more, see Algorithm characterizations.

See also

- Abstract machine
- Algorithm characterizations
- Algorithm design
- Algorithmic efficiency
- Algorithm engineering
- Algorithm examples
- Algorithmic music
- Garbage In, Garbage Out
- Algorithmic synthesis

- Algorithmic trading
- Data structure
- Heuristics
- *Introduction to Algorithms*
- Important algorithm-related publications
- List of algorithm general topics
- List of algorithms
- List of terms relating to algorithms and data structures
- Partial function
- Profiling (computer programming)
- Program optimization
- Theory of computation
  - Computability (part of computability theory)
  - Computational complexity theory
- Randomized algorithm and quantum algorithm

References

- Axt, P. (1959) On a Subrecursive Hierarchy and Primitive Recursive Degrees, *Transactions of the American Mathematical Society* 92, pp. 85–105
- Bell, C. Gordon and Newell, Allen (1971), *Computer Structures: Readings and Examples*, McGraw-Hill Book Company, New York. ISBN 0070043574}.
- Blass, Andreas; Gurevich, Yuri (2003). "Algorithms: A Quest for Absolute Definitions" [47]. *Bulletin of European Association for Theoretical Computer Science* **81**. Includes an excellent bibliography of 56 references.
- Boolos, George; Jeffrey, Richard (1974, 1980, 1989, 1999). *Computability and Logic* (4th ed.). Cambridge University Press, London. ISBN 0-521-20402-X.: cf. Chapter 3 *Turing machines* where they discuss "certain enumerable sets not effectively (mechanically) enumerable".
- Burgin, M. *Super-recursive algorithms*, Monographs in computer science, Springer, 2005. ISBN 0387955690
- Campagnolo, M.L., Moore, C., and Costa, J.F. (2000) An analog characterization of the subrecursive functions. In *Proc. of the 4th Conference on Real Numbers and Computers*, Odense University, pp. 91–109
- Church, Alonzo (1936a). "An Unsolvable Problem of Elementary Number Theory". *The American Journal of Mathematics* **58** (2): 345–363. doi:10.2307/2371045. Reprinted in *The Undecidable*, p. 89ff. The first expression of "Church's Thesis". See in particular page 100 (*The Undecidable*) where he defines the notion of "effective calculability" in terms of "an algorithm", and he uses the word "terminates", etc.
- Church, Alonzo (1936b). "A Note on the Entscheidungsproblem". *The Journal of Symbolic Logic* **1** (1): 40–41. doi:10.2307/2269326. JSTOR 2269326. Church, Alonzo (1936). "Correction to a Note on the Entscheidungsproblem". *The Journal of Symbolic Logic* **1** (3): 101–102. doi:10.2307/2269030. JSTOR 2269030. Reprinted in *The Undecidable*, p. 110ff. Church shows that the Entscheidungsproblem is unsolvable in about 3 pages of text and 3 pages of footnotes.
- Daffa', Ali Abdullah al- (1977). *The Muslim contribution to mathematics*. London: Croom Helm. ISBN 0-85664-464-1.
- Davis, Martin (1965). *The Undecidable: Basic Papers On Undecidable Propositions, Unsolvable Problems and Computable Functions*. New York: Raven Press. ISBN 0486432289. Davis gives commentary before each article. Papers of Gödel, Alonzo Church, Turing, Rosser, Kleene, and Emil Post are included; those cited in the article are listed here by author's name.
- Davis, Martin (2000). *Engines of Logic: Mathematicians and the Origin of the Computer*. New York: W. W. Nortion. ISBN 0393322297. Davis offers concise biographies of Leibniz, Boole, Frege, Cantor, Hilbert, Gödel and Turing with von Neumann as the show-stealing villain. Very brief bios of Joseph-Marie Jacquard, Babbage,

Ada Lovelace, Claude Shannon, Howard Aiken, etc.

- *This article incorporates public domain material from the NIST* document "algorithm" [48] by Paul E. Black (Dictionary of Algorithms and Data Structures).
- Dennett, Daniel (1995). *Darwin's Dangerous Idea*. New York: Touchstone/Simon & Schuster. ISBN 0684802902.
- Yuri Gurevich, *Sequential Abstract State Machines Capture Sequential Algorithms* [49], ACM Transactions on Computational Logic, Vol 1, no 1 (July 2000), pages 77–111. Includes bibliography of 33 sources.
- Kleene C., Stephen (1936). "General Recursive Functions of Natural Numbers". *Mathematische Annalen* **112** (5): 727–742. doi:10.1007/BF01565439. Presented to the American Mathematical Society, September 1935. Reprinted in *The Undecidable*, p. 237ff. Kleene's definition of "general recursion" (known now as mu-recursion) was used by Church in his 1935 paper *An Unsolvable Problem of Elementary Number Theory* that proved the "decision problem" to be "undecidable" (i.e., a negative result).
- Kleene C., Stephen (1943). "Recursive Predicates and Quantifiers". *American Mathematical Society Transactions* **54** (1): 41–73. doi:10.2307/1990131. Reprinted in *The Undecidable*, p. 255ff. Kleene refined his definition of "general recursion" and proceeded in his chapter "12. Algorithmic theories" to posit "Thesis I" (p. 274); he would later repeat this thesis (in Kleene 1952:300) and name it "Church's Thesis"(Kleene 1952:317) (i.e., the Church thesis).
- Kleene, Stephen C. (First Edition 1952). *Introduction to Metamathematics* (Tenth Edition 1991 ed.). North-Holland Publishing Company. ISBN 0720421039. Excellent — accessible, readable — reference source for mathematical "foundations".
- Knuth, Donald (1997). *Fundamental Algorithms, Third Edition*. Reading, Massachusetts: Addison–Wesley. ISBN 0201896834.
- Kosovsky, N. K. *Elements of Mathematical Logic and its Application to the theory of Subrecursive Algorithms*, LSU Publ., Leningrad, 1981
- Kowalski, Robert (1979). "Algorithm=Logic+Control". *Communications of the ACM* **22** (7): 424–436. doi:10.1145/359131.359136. ISSN 0001-0782.
- A. A. Markov (1954) *Theory of algorithms*. [Translated by Jacques J. Schorr-Kon and PST staff] Imprint Moscow, Academy of Sciences of the USSR, 1954 [i.e., Jerusalem, Israel Program for Scientific Translations, 1961; available from the Office of Technical Services, U.S. Dept. of Commerce, Washington] Description 444 p. 28 cm. Added t.p. in Russian Translation of Works of the Mathematical Institute, Academy of Sciences of the USSR, v. 42. Original title: Teoriya algerifmov. [QA248.M2943 Dartmouth College library. U.S. Dept. of Commerce, Office of Technical Services, number OTS 60-51085.]
- Minsky, Marvin (1967). *Computation: Finite and Infinite Machines* (First ed.). Prentice-Hall, Englewood Cliffs, NJ. ISBN 0131654497. Minsky expands his "...idea of an algorithm — an effective procedure..." in chapter 5.1 *Computability, Effective Procedures and Algorithms. Infinite machines."*
- Post, Emil (1936). "Finite Combinatory Processes, Formulation I". *The Journal of Symbolic Logic* **1** (3): 103–105. doi:10.2307/2269031. Reprinted in *The Undecidable*, p. 289ff. Post defines a simple algorithmic-like process of a man writing marks or erasing marks and going from box to box and eventually halting, as he follows a list of simple instructions. This is cited by Kleene as one source of his "Thesis I", the so-called Church–Turing thesis.
- Rosser, J.B. (1939). "An Informal Exposition of Proofs of Godel's Theorem and Church's Theorem". *Journal of Symbolic Logic* **4**. Reprinted in *The Undecidable*, p. 223ff. Herein is Rosser's famous definition of "effective method": "...a method each step of which is precisely predetermined and which is certain to produce the answer in a finite number of steps... a machine which will then solve any problem of the set with no human intervention beyond inserting the question and (later) reading the answer" (p. 225–226, *The Undecidable*)
- Sipser, Michael (2006). *Introduction to the Theory of Computation*. PWS Publishing Company. ISBN 053494728X.

- Stone, Harold S. (1972). *Introduction to Computer Organization and Data Structures* (1972 ed.). McGraw-Hill, New York. ISBN 0070617260. Cf. in particular the first chapter titled: *Algorithms, Turing Machines, and Programs*. His succinct informal definition: "...any sequence of instructions that can be obeyed by a robot, is called an *algorithm*" (p. 4).
- Turing, Alan M. (1936–7). "On Computable Numbers, With An Application to the Entscheidungsproblem". *Proceedings of the London Mathematical Society, Series 2* **42**: 230–265. doi:10.1112/plms/s2-42.1.230.. Corrections, ibid, vol. 43(1937) pp. 544–546. Reprinted in *The Undecidable*, p. 116ff. Turing's famous paper completed as a Master's dissertation while at King's College Cambridge UK.
- Turing, Alan M. (1939). "Systems of Logic Based on Ordinals". *Proceedings of the London Mathematical Society, Series 2* **45**: 161–228. doi:10.1112/plms/s2-45.1.161. Reprinted in *The Undecidable*, p. 155ff. Turing's paper that defined "the oracle" was his PhD thesis while at Princeton USA.
- United States Patent and Trademark Office (2006), *2106.02 **>Mathematical Algorithms< - 2100 Patentability* [50], Manual of Patent Examining Procedure (MPEP). Latest revision August 2006

Secondary references

- Bolter, David J. (1984). *Turing's Man: Western Culture in the Computer Age* (1984 ed.). The University of North Carolina Press, Chapel Hill NC. ISBN 0807815640., ISBN 0-8078-4108-0 pbk.
- Dilson, Jesse (2007). *The Abacus* ((1968,1994) ed.). St. Martin's Press, NY. ISBN 031210409X., ISBN 0-312-10409-X (pbk.)
- van Heijenoort, Jean (2001). *From Frege to Gödel, A Source Book in Mathematical Logic, 1879–1931* ((1967) ed.). Harvard University Press, Cambridge, MA. ISBN 0674324498., 3rd edition 1976[?], ISBN 0-674-32449-8 (pbk.)
- Hodges, Andrew (1983). *Alan Turing: The Enigma* ((1983) ed.). Simon and Schuster, New York. ISBN 0671492071., ISBN 0-671-49207-1. Cf. Chapter "The Spirit of Truth" for a history leading to, and a discussion of, his proof.

Further reading

- David Harel, Yishai A. Feldman, *Algorithmics: the spirit of computing*, Edition 3, Pearson Education, 2004, ISBN 0321117840
- Jean-Luc Chabert, Évelyne Barbin, *A history of algorithms: from the pebble to the microchip*, Springer, 1999, ISBN 3540633693

External links

- The Stony Brook Algorithm Repository [51]
- Weisstein, Eric W., "Algorithm [52]" from MathWorld.
- Algorithms in Everyday Mathematics [53]
- Algorithms [54] at the Open Directory Project
- Sortier- und Suchalgorithmen (German) [55]
- Jeff Erickson Algorithms course material [56]

References

[1] Kleene 1943 in Davis 1965:274

[2] Rosser 1939 in Davis 1965:225

[3] Adaptation and learning in automatic systems (http://books.google.com/books?id=sgDHJlafMskC), page 54, Ya. Z. Tsypkin, Z. J. Nikolic, Academic Press, 1971, ISBN 9780127020501

[4] Boolos and Jeffrey 1974,1999:19

[5] Minsky 1967:105

[6] Gurevich 2000:1, 3

[7] Kleene 1952:136

[8] Knuth 1997:5

[9] Boldface added, Kleene 1952:137

[10] Kleene 1952:325

[11] Davis 1958:12–15

[12] Kleene 1952:332

[13] Minsky 1967:186

[14] Sipser 2006:157

[15] Kleene 1952:229 shows that "Definition by cases" is primitive recursive. CASES requires that the list of testable instances within the CASE definition to be (i) mutually exclusive and (ii) collectively exhaustive i.e. it must include or "cover" all possibility. The CASE statement proceeds in numerical order and exits at the first successful test; see more at Boolos–Burgess–Jeffrey Fourth edition 2002:74

[16] An IF-THEN-ELSE or "logical test with branching" is just a CASE instruction reduced to two outcomes: (i) test is successful, (ii) test is unsuccessful. The IF-THEN-ELSE is closely related to the AND-OR-INVERT logic function from which all 16 logical "operators" of one or two variables can be derived; see more at Propositional formula. Like definition by cases, a sequence of IF-THEN-ELSE logical tests must be mutually exclusive and collectively exhaustive over the variables tested.

[17] Kowalski 1979

[18] Sue Carroll, Taz Daughtrey. *Fundamental Concepts for the Software Quality Engineer* (http://books.google.com/books?id=bz_cl3B05IcC&pg=PA282). pp. 282 et seq.. .

[19] Davis 2000:18

[20] Bolter 1984:24

[21] Bolter 1984:26

[22] Bolter 1984:33–34, 204–206)

[23] All quotes from W. Stanley Jevons 1880 *Elementary Lessons in Logic: Deductive and Inductive*, Macmillan and Co., London and New York. Republished as a googlebook; cf Jevons 1880:199–201. Louis Couturat 1914 *the Algebra of Logic*, The Open Court Publishing Company, Chicago and London. Republished as a googlebook; cf Couturat 1914:75–76 gives a few more details; interestingly he compares this to a typewriter as well as a piano. Jevons states that the account is to be found at Jan. 20, 1870 *The Proceedings of the Royal Society*.

[24] Jevons 1880:199–200

[25] All quotes from John Venn 1881 *Symbolic Logic*, Macmillan and Co., London. Republished as a googlebook. cf Venn 1881:120–125. The interested reader can find a deeper explanation in those pages.

[26] Bell and Newell diagram 1971:39, cf. Davis 2000

[27] * Melina Hill, Valley News Correspondent, *A Tinkerer Gets a Place in History*, Valley News West Lebanon NH, Thursday March 31, 1983, page 13.

[28] Davis 2000:14

[29] van Heijenoort 1967:81ff

[30] van Heijenoort's commentary on Frege's *Begriffsschrift, a formula language, modeled upon that of arithmetic, for pure thought* in van Heijenoort 1967:1

[31] Dixon 1906, cf. Kleene 1952:36–40

[32] cf. footnote in Alonzo Church 1936a in Davis 1965:90 and 1936b in Davis 1965:110

[33] Kleene 1935–6 in Davis 1965:237ff, Kleene 1943 in Davis 1965:255ff

[34] Church 1936 in Davis 1965:88ff

[35] cf. "Formulation I", Post 1936 in Davis 1965:289–290

[36] Turing 1936–7 in Davis 1965:116ff

[37] Rosser 1939 in Davis 1965:226

[38] Kleene 1943 in Davis 1965:273–274

[39] Kleene 1952:300, 317

[40] Kleene 1952:376

[41] Turing 1936–7 in Davis 1965:289–290

[42] Turing 1936 in Davis 1965, Turing 1939 in Davis 1965:160

[43] Hodges, p. 96

[44] Turing 1936–7:116)

[45] Turing 1936–7 in Davis 1965:136
[46] Turing 1939 in Davis 1965:160
[47] http://research.microsoft.com/~gurevich/Opera/164.pdf
[48] http://www.nist.gov/dads/HTML/algorithm.html
[49] http://research.microsoft.com/~gurevich/Opera/141.pdf
[50] http://www.uspto.gov/web/offices/pac/mpep/documents/2100_2106_02.htm
[51] http://www.cs.sunysb.edu/~algorith/
[52] http://mathworld.wolfram.com/Algorithm.html
[53] http://everydaymath.uchicago.edu/educators/Algorithms_final.pdf
[54] http://www.dmoz.org/Computers/Algorithms//
[55] http://sortieralgorithmen.de/
[56] http://compgeom.cs.uiuc.edu/~jeffe//teaching/algorithms/

Approximation algorithm

In computer science and operations research, **approximation algorithms** are algorithms used to find approximate solutions to optimization problems. Approximation algorithms are often associated with NP-hard problems; since it is unlikely that there can ever be efficient polynomial time exact algorithms solving NP-hard problems, one settles for polynomial time sub-optimal solutions. Unlike heuristics, which usually only find reasonably good solutions reasonably fast, one wants provable solution quality and provable run time bounds. Ideally, the approximation is optimal up to a small constant factor (for instance within 5% of the optimal solution). Approximation algorithms are increasingly being used for problems where exact polynomial-time algorithms are known but are too expensive due to the input size.

A typical example for an approximation algorithm is the one for vertex cover in graphs: find an uncovered edge and add *both* endpoints to the vertex cover, until none remain. It is clear that the resulting cover is at most twice as large as the optimal one. This is a constant factor approximation algorithm with a factor of 2.

NP-hard problems vary greatly in their approximability; some, such as the bin packing problem, can be approximated within any factor greater than 1 (such a family of approximation algorithms is often called a polynomial time approximation scheme or *PTAS*). Others are impossible to approximate within any constant, or even polynomial factor unless P = NP, such as the maximum clique problem.

NP-hard problems can often be expressed as integer programs (IP) and solved exactly in exponential time. Many approximation algorithms emerge from the linear programming relaxation of the integer program.

Not all approximation algorithms are suitable for all practical applications. They often use IP/LP/Semidefinite solvers, complex data structures or sophisticated algorithmic techniques which lead to difficult implementation problems. Also, some approximation algorithms have impractical running times even though they are polynomial time, for example $O(n^{2000})$. Yet the study of even very expensive algorithms is not a completely theoretical pursuit as they can yield valuable insights. A classic example is the initial PTAS for Euclidean TSP due to Sanjeev Arora which had prohibitive running time, yet within a year, Arora refined the ideas into a linear time algorithm. Such algorithms are also worthwhile in some applications where the running times and cost can be justified e.g. computational biology, financial engineering, transportation planning, and inventory management. In such scenarios, they must compete with the corresponding direct IP formulations.

Another limitation of the approach is that it applies only to optimization problems and not to "pure" decision problems like satisfiability, although it is often possible to conceive optimization versions of such problems, such as the maximum satisfiability problem (Max SAT).

Inapproximability has been a fruitful area of research in computational complexity theory since the 1990 result of Feige, Goldwasser, Lovasz, Safra and Szegedy on the inapproximability of Independent Set. After Arora et al. proved the PCP theorem a year later, it has now been shown that Johnson's 1974 approximation algorithms for Max SAT, Set Cover, Independent Set and Coloring all achieve the optimal approximation ratio, assuming P != NP.

Performance guarantees

For some approximation algorithms it is possible to prove certain properties about the approximation of the optimum result. For example, in the case of a **ϱ-approximation algorithm** *A* it has been proven that the value/cost, *f*(*x*), of the approximate solution *A*(*x*) to an instance *x* will not be more (or less, depending on the situation) than a factor ϱ times the value, OPT, of an optimum solution.

$$\begin{cases} \mathrm{OPT} \leq f(x) \leq \rho \mathrm{OPT}, & \text{if } \rho > 1; \\ \rho \mathrm{OPT} \leq f(x) \leq \mathrm{OPT}, & \text{if } \rho < 1. \end{cases}$$

The factor ϱ is called the *relative performance guarantee*. An approximation algorithm has an *absolute performance guarantee* or *bounded error c*, if it has been proven for every instance *x* that

$$(\mathrm{OPT} - c) \leq f(x) \leq (\mathrm{OPT} + c).$$

Similarly, the *performance guarantee*, *R*(*x*,*y*), of a solution *y* to an instance *x* is defined as

$$\mathrm{R(x,y)} = \max\left(\frac{OPT}{f(y)}, \frac{f(y)}{OPT}\right),$$

where *f*(*y*) is the value/cost of the solution *y* for the instance *x*. Clearly, the performance guarantee is greater than or equal to 1 and equal to 1 if and only if *y* is an optimal solution. If an algorithm *A* guarantees to return solutions with a performance guarantee of at most *r*(*n*), then *A* is said to be an *r*(*n*)-approximation algorithm and has an *approximation ratio* of *r*(*n*). Likewise, a problem with an *r*(*n*)-approximation algorithm is said to be r(n)*-approximable* or have an approximation ratio of *r*(*n*).[1] [2]

One may note that for minimization problems, the two different guarantees provide the same result and that for maximization problems, a relative performance guarantee of ρ is equivalent to a performance guarantee of $r = \rho^{-1}$. In the literature, both definitions are common but it is clear which definition is used since, for maximization problems, as ρ ≤ 1 while r ≥ 1.

The *absolute performance guarantee* P_A of some approximation algorithm *A*, where *x* refers to an instance of a problem, and where $R_A(x)$ is the performance guarantee of *A* on *x* (i.e. ρ for problem instance *x*) is:

$$\mathrm{P}_A = \inf\{r \geq 1 \mid R_A(I) \leq r, \forall x\}$$

That is to say that P_A is the largest bound on the approximation ratio, *r*, that one sees over all possible instances of the problem. Likewise, the *asymptotic performance ratio* R_A^∞ is:

$$R_A^\infty = \inf\{r \geq 1 \mid \exists n \in \mathbb{Z}^+, R_A(I) \leq r, \forall x, |x| \geq n\}$$

That is to say that it is the same as the *absolute performance ratio*, with a lower bound *n* on the size of problem instances. These two types of ratios are used because there exist algorithms where the difference between these two is significant.

Performance guarantees

	***r*-approx**[1] [2]	**ϱ-approx**	**rel. error**[2]	**rel. error**[1]	**norm. rel. error**[1] [2]	**abs. error**[1] [2]
max: $f(x) \geq$	$r^{-1}\mathrm{OPT}$	$\rho\mathrm{OPT}$	$(1-c)\mathrm{OPT}$	$(1-c)\mathrm{OPT}$	$(1-c)\mathrm{OPT} + c\mathrm{WORST}$	$\mathrm{OPT} - c$
min: $f(x) \leq$	$r\mathrm{OPT}$	$\rho\mathrm{OPT}$	$(1+c)\mathrm{OPT}$	$(1-c)^{-1}\mathrm{OPT}$	$(1-c)^{-1}\mathrm{OPT} + c\mathrm{WORST}$	$\mathrm{OPT} + c$

Epsilon terms

In the literature, an approximation ratio for a maximization (minimization) problem of $c - \epsilon$ (min: $c + \epsilon$) means that the algorithm has an approximation ratio of $c \mp \epsilon$ for arbitrary $\epsilon > 0$ but that the ratio has not (or cannot) be shown for $\epsilon = 0$. An example of this is the optimal inapproximability — inexistence of approximation — ratio of $7 / 8 + \epsilon$ for satisfiable MAX-3SAT instances due to Johan Håstad.[3] As mentioned previously, when $c = 1$, the problem is said to have a polynomial-time approximation scheme.

An ϵ-term may appear when an approximation algorithm introduces a multiplicative error and a constant error while the minimum optimum of instances of size n goes to infinity as n does. In this case, the approximation ratio is $c \mp k / \mathrm{OPT} = c \mp o(1)$ for some constants c and k. Given arbitrary $\epsilon > 0$, one can choose a large enough N such that the term $k / \mathrm{OPT} < \epsilon$ for every $n \geq N$. For every fixed ϵ, instances of size $n < N$ can be solved by brute force , thereby showing an approximation ratio — existence of approximation algorithms with a guarantee — of $c \mp \epsilon$ for every $\epsilon > 0$.

See also

- Domination analysis considers guarantees in terms of the rank of the computed solution.

References

[1] G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, and M. Protasi (1999). *Complexity and Approximation: Combinatorial Optimization Problems and their Approximability Properties*.

[2] Viggo Kann (1992). *On the Approximability of NP-complete Optimization Problems* (http://www.csc.kth.se/~viggo/papers/phdthesis.pdf). .

[3] Johan Håstad (1999). "Some Optimal Inapproximability Results" (http://www.nada.kth.se/~johanh/optimalinap.ps). *Journal of the ACM*. .

- Vazirani, Vijay V. (2003). *Approximation Algorithms*. Berlin: Springer. ISBN 3540653678.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Chapter 35: Approximation Algorithms, pp. 1022–1056.
- Dorit H. Hochbaum, ed. *Approximation Algorithms for NP-Hard problems, PWS Publishing Company, 1997. ISBN 0-534-94968-1. Chapter 9: Various Notions of Approximations: Good, Better, Best, and More*

External links

- Pierluigi Crescenzi, Viggo Kann, Magnús Halldórsson, Marek Karpinski and Gerhard Woeginger, *A compendium of NP optimization problems* (http://www.nada.kth.se/~viggo/wwwcompendium/).

Search algorithm

In computer science, a **search algorithm**, broadly speaking, is an algorithm for finding an item with specified properties among a collection of items. The items may be stored individually as records in a database; or may be elements of a search space defined by a mathematical formula or procedure, such as the roots of an equation with integer variables; or a combination of the two, such as the Hamiltonian circuits of a graph.

Classes of search algorithms

For explicitly stored databases

Algorithms for searching in explicitly stored databases include the simple linear search, and many other algorithms that use a variety of search data structures, such as binary search trees, heaps and hash tables, to speed up multiple queries over the same database.

There are also many algorithms designed specifically for retrieval in very large databases, such as bank account records, electronic documents, product catalogs, fingerprint and image databases, and so on.

For virtual search spaces

Algorithms for searching virtual spaces are used in constraint satisfaction problem, where the goal is to find a set of value assignments to certain variables that will satisfy specific mathematical equations and inequations. They are also used when the goal is to find a variable assignment that will maximize or minimize a certain function of those variables. Algorithms for these problems include the basic brute-force search (also called "naïve" or "uninformed" search), and a variety of heuristics that try to exploit partial knowledge about structure of the space, such as linear relaxation, constraint generation, and constraint propagation.

An important subclass are the Local search methods, that view the elements of the search space as the vertices of a graph, with edges defined by a set of heuristics applicable to the case; and scan the space by moving from item to item along the edges, for example according to the steepest descent or best-first criterion, or in a stochastic search. This category includes a great variety of general metaheuristic methods, such as simulated annealing, tabu search, A-teams, and genetic programming, that combine arbitrary heuristics in specific ways. Examples of these methods are the and methods.

This class also includes various tree search algorithms, that view the elements as vertices of a tree, and traverse that tree in some special order. Examples of the latter include the exhaustive methods such as depth-first search and breadth-first search, as well as various heuristic-based search tree pruning methods such as backtracking and branch and bound. Unlike general metaheuristics, which at best work only in a probabilistic sense, many of these tree-search methods are guaranteed to find the exact or optimal solution, if given enough time.

Another important sub-class consists of algorithms for exploring the game tree of multiple-player games, such as chess or backgammon, whose nodes consist of all possible game situations that could result from the current situation. The goal in these problems is to find the move that provides the best chance of a win, taking into account all possible moves of the opponent(s). Similar problems occur when humans or machines have to make successive decisions whose outcomes are not entirely under one's control, such as in robot guidance or in marketing, financial or military strategy planning. This kind of problems has been extensively studied in the context of artificial intelligence. Examples of algorithms for this class are the minimax algorithm, alpha-beta pruning, and the A* algorithm.

For sub-structures of a given structure

The name combinatorial search is generally used for algorithms that look for a specific sub-structure of a given discrete structure, such as a graph, a string, a finite group, and so on. The term combinatorial optimization is typically used when the goal is to find a sub-structure with a maximum (or minimum) value of some parameter. (Since the sub-structure is usually represented in the computer by a set of integer variables with constraints, these problems can be viewed as special cases of constraint satisfaction or discrete optimization; but they are usually formulated and solved in a more abstract setting where the internal representation is not explicitly mentioned.)

An important and extensively studied subclass are the graph algorithms, in particular graph traversal algorithms, for finding specific sub-structures in a given graph — such as subgraphs, paths, circuits, and so on. Examples include Dijkstra's algorithm, Kruskal's algorithm, the nearest neighbour algorithm, and Prim's algorithm.

Another important subclass of this category are the string searching algorithms, that search for patterns within strings. Two famous examples are the Boyer–Moore and Knuth–Morris–Pratt algorithms, and several algorithms based on the suffix tree data structure.

For quantum computers

There are also search methods designed for (currently non-existent) quantum computers, like Grover's algorithm, that are theoretically faster than linear or brute-force search even without the help of data structures or heuristics.

See also

- Search games
- Recommender systems also use statistical methods to rank results in very large data sets
- Sorting algorithms necessary for executing certain search algorithms
- Selection algorithm
- No free lunch in search and optimization
- Search engine (computing)
- Linear search problem

References

- Donald Knuth. *The Art of Computer Programming. Volume 3: Sorting and Searching*. ISBN 0-201-89685-0.

External links

- Uninformed Search Project [1] at the Wikiversity.
- Unsorted Data Searching Using Modulated Database [2].

References

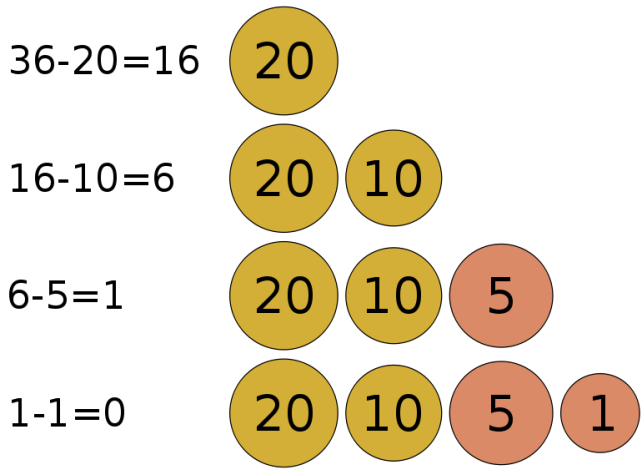
[1] http://en.wikiversity.org/wiki/Uninformed_Search_Project

[2] http://sites.google.com/site/hantarto/quantum-computing/unsorted

Greedy algorithm

A **greedy algorithm** is any algorithm that follows the problem solving metaheuristic of making the locally optimal choice at each stage[1] with the hope of finding the global optimum.

For example, applying the greedy strategy to the traveling salesman problem yields the following algorithm: "At each stage visit the unvisited city nearest to the current city".



The greedy algorithm determines the minimum number of coins to give while making change. These are the steps a human would take to emulate a greedy algorithm to represent 36 cents using coins. The coin of the highest value, less than the remaining change owed, is the local optimum. (Note that in general the change-making problem requires dynamic programming or integer programming to find an optimal solution; US and other currencies are special cases where the greedy strategy works.)

Specifics

In general, greedy algorithms have five pillars:

1. A candidate set, from which a solution is created
2. A selection function, which chooses the best candidate to be added to the solution
3. A feasibility function, that is used to determine if a candidate can be used to contribute to a solution
4. An objective function, which assigns a value to a solution, or a partial solution, and
5. A solution function, which will indicate when we have discovered a complete solution

Greedy algorithms produce good solutions on some mathematical problems, but not on others. Most problems for which they work well have two properties:

Greedy choice property

> We can make whatever choice seems best at the moment and then solve the subproblems that arise later. The choice made by a greedy algorithm may depend on choices made so far but not on future choices or all the solutions to the subproblem. It iteratively makes one greedy choice after another, reducing each given problem into a smaller one. In other words, a greedy algorithm never reconsiders its choices. This is the main difference from dynamic programming, which is exhaustive and is guaranteed to find the solution. After every stage, dynamic programming makes decisions based on all the decisions made in the previous stage, and may reconsider the previous stage's algorithmic path to solution.

Optimal substructure

> "A problem exhibits optimal substructure if an optimal solution to the problem contains optimal solutions to the sub-problems."[2] Said differently, a problem has optimal substructure if the best next move always leads to the optimal solution. An example of 'non-optimal substructure' would be a situation where capturing a queen in chess (good next move) will eventually lead to the loss of the game (bad overall move).

When greedy-type algorithms fail

For many other problems, greedy algorithms fail to produce the optimal solution, and may even produce the *unique worst possible* solutions. One example is the nearest neighbor algorithm mentioned above: for each number of cities there is an assignment of distances between the cities for which the nearest neighbor heuristic produces the unique worst possible tour.[3]

Imagine the coin example with only 25-cent, 10-cent, and 4-cent coins. The greedy algorithm would not be able to make change for 41 cents, since after committing to use one 25-cent coin and one 10-cent coin it would be impossible to use 4-cent coins for the balance of 6 cents. Whereas a person or a more sophisticated algorithm could make change for 41 cents change with one 25-cent coin and four 4-cent coins.

Types

Greedy algorithms can be characterized as being 'short sighted', and as 'non-recoverable'. They are ideal only for problems which have 'optimal substructure'. Despite this, greedy algorithms are best suited for simple problems (e.g. giving change). It is important, however, to note that the greedy algorithm can be used as a selection algorithm to prioritize options within a search, or branch and bound algorithm. There are a few variations to the greedy algorithm:

- Pure greedy algorithms
- Orthogonal greedy algorithms
- Relaxed greedy algorithms

Applications

Greedy algorithms mostly (but not always) fail to find the globally optimal solution, because they usually do not operate exhaustively on all the data. They can make commitments to certain choices too early which prevent them from finding the best overall solution later. For example, all known greedy coloring algorithms for the graph coloring problem and all other NP-complete problems do not consistently find optimum solutions. Nevertheless, they are useful because they are quick to think up and often give good approximations to the optimum.

If a greedy algorithm can be proven to yield the global optimum for a given problem class, it typically becomes the method of choice because it is faster than other optimisation methods like dynamic programming. Examples of such greedy algorithms are Kruskal's algorithm and Prim's algorithm for finding minimum spanning trees, Dijkstra's algorithm for finding single-source shortest paths, and the algorithm for finding optimum Huffman trees.

The theory of matroids, and the more general theory of greedoids, provide whole classes of such algorithms.

Greedy algorithms appear in network routing as well. Using greedy routing, a message is forwarded to the neighboring node which is "closest" to the destination. The notion of a node's location (and hence "closeness") may be determined by its physical location, as in geographic routing used by ad-hoc networks. Location may also be an entirely artificial construct as in small world routing and distributed hash table.

Examples

- In the Macintosh computer game Crystal Quest the objective is to collect crystals, in a fashion similar to the travelling salesman problem. The game has a demo mode, where the game uses a greedy algorithm to go to every crystal. Unfortunately, the artificial intelligence does not account for obstacles, so the demo mode often ends quickly.
- The Matching pursuit is an example of greedy algorithm applied on signal approximation.

See also

- Greedy source
- Matroid
- Epsilon-greedy strategy

References

- *Introduction to Algorithms* (Cormen, Leiserson, and Rivest) 1990, Chapter 16 "Greedy Algorithms" p. 329.
- *Introduction to Algorithms* (Cormen, Leiserson, Rivest, and Stein) 2001, Chapter 16 "Greedy Algorithms".
- G. Gutin, A. Yeo and A. Zverovich, Traveling salesman should not be greedy: domination analysis of greedy-type heuristics for the TSP. Discrete Applied Mathematics 117 (2002), 81–86.
- J. Bang-Jensen, G. Gutin and A. Yeo, When the greedy algorithm fails. Discrete Optimization 1 (2004), 121–127.
- G. Bendall and F. Margot, Greedy Type Resistance of Combinatorial Problems, Discrete Optimization 3 (2006), 288–298.

External links

- Greedy algorithm visualization [4] A visualization of a greedy solution to the N-Queens puzzle by Yuval Baror.

References

[1] Paul E. Black, "greedy algorithm" in *Dictionary of Algorithms and Data Structures* [online], U.S. National Institute of Standards and Technology, February 2005, webpage: NIST-greedyalgo (http://www.nist.gov/dads/HTML/greedyalgo.html).

[2] Introduction to Algorithms (Cormen, Leiserson, Rivest, and Stein) 2001, Chapter 16 "Greedy Algorithms".

[3] (G. Gutin, A. Yeo and A. Zverovich, 2002)

[4] http://yuval.bar-or.org/index.php?item=9

Divide and conquer algorithm

In computer science, **divide and conquer** (**D&C**) is an important algorithm design paradigm based on multi-branched recursion. A divide and conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same (or related) type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem.

This technique is the basis of efficient algorithms for all kinds of problems, such as sorting (e.g., quicksort, merge sort), multiplying large numbers (e.g. Karatsuba), syntactic analysis (e.g., top-down parsers), and computing the discrete Fourier transform (FFTs).

On the other hand, the ability to understand and design D&C algorithms is a skill that takes time to master. As when proving a theorem by induction, it is often necessary to replace the original problem by a more general or complicated problem in order to get the recursion going, and there is no systematic method for finding the proper generalization.

The name "divide and conquer" is sometimes applied also to algorithms that reduce each problem to only one subproblem, such as the binary search algorithm for finding a record in a sorted list (or its analog in numerical computing, the bisection algorithm for root finding) [1] . These algorithms can be implemented more efficiently than general divide-and-conquer algorithms; in particular, if they use tail recursion, they can be converted into simple loops. Under this broad definition, however, every algorithm that uses recursion or loops could be regarded as a "divide and conquer algorithm". Therefore, some authors consider that the name "divide and conquer" should be used only when each problem may generate two or more subproblems.[2] The name **decrease and conquer** has been proposed instead for the single-subproblem class.[3]

The correctness of a divide and conquer algorithm is usually proved by mathematical induction, and its computational cost is often determined by solving recurrence relations.

Early historical examples

Binary search, a divide and conquer algorithm in which the original problem is successively broken down into *single* subproblems of roughly half the original size, has a long history. The idea of using a sorted list of items to facilitate searching dates back as far as Babylonia in 200BC,[4] while a clear description of the algorithm on computers appeared in 1946 in an article by John Mauchly.[4] Another divide and conquer algorithm with a single subproblem is the Euclidean algorithm to compute the greatest common divisor of two numbers (by reducing the numbers to smaller and smaller equivalent subproblems), which dates to several centuries BC.

An early example of a divide-and-conquer algorithm with multiple subproblems is Gauss's 1805 description of what is now called the Cooley-Tukey fast Fourier transform (FFT) algorithm,[5] although he did not analyze its operation count quantitatively and FFTs did not become widespread until they were rediscovered over a century later.

An early two-subproblem D&C algorithm that was specifically developed for computers and properly analyzed is the merge sort algorithm, invented by John von Neumann in 1945.[6]

Another notable example is the algorithm invented by Anatolii A. Karatsuba in 1960[7] that could multiply two *n*-digit numbers in $O(n^{\log_2 3})$operations. This algorithm disproved Andrey Kolmogorov's 1956 conjecture that $\Omega(n^2)$operations would be required for that task.

As another example of a divide and conquer algorithm that did not originally involve computers, Knuth gives the method a post office typically uses to route mail: letters are sorted into separate bags for different geographical areas, each of these bags is itself sorted into batches for smaller sub-regions, and so on until they are delivered.[4] This is related to a radix sort, described for punch-card sorting machines as early as 1929.[4]

Advantages

Solving difficult problems

Divide and conquer is a powerful tool for solving conceptually difficult problems, such as the classic Tower of Hanoi puzzle: all it requires is a way of breaking the problem into sub-problems, of solving the trivial cases and of combining sub-problems to the original problem.

Algorithm efficiency

The divide-and-conquer paradigm often helps in the discovery of efficient algorithms. It was the key, for example, to Karatsuba's fast multiplication method, the quicksort and mergesort algorithms, the Strassen algorithm for matrix multiplication, and fast Fourier transforms.

In all these examples, the D&C approach led to an improvement in the asymptotic cost of the solution. For example, if the base cases have constant-bounded size, the work of splitting the problem and combining the partial solutions is proportional to the problem's size n, and there are a bounded number p of subproblems of size ~ n/p at each stage, then the cost of the divide-and-conquer algorithm will be $O(n \log n)$.

Parallelism

Divide and conquer algorithms are naturally adapted for execution in multi-processor machines, especially shared-memory systems where the communication of data between processors does not need to be planned in advance, because distinct sub-problems can be executed on different processors.

Memory access

Divide-and-conquer algorithms naturally tend to make efficient use of memory caches. The reason is that once a sub-problem is small enough, it and all its sub-problems can, in principle, be solved within the cache, without accessing the slower main memory. An algorithm designed to exploit the cache in this way is called *cache oblivious*, because it does not contain the cache size(s) as an explicit parameter.[8] Moreover, D&C algorithms can be designed for many important algorithms, such as sorting, FFTs, and matrix multiplication, in such a way as to be *optimal cache oblivious* algorithms—they use the cache in a provably optimal way, in an asymptotic sense, regardless of the cache size. In contrast, the traditional approach to exploiting the cache is *blocking*, where the problem is explicitly divided into chunks of the appropriate size—this can also use the cache optimally, but only when the algorithm is tuned for the specific cache size(s) of a particular machine.

The same advantage exists with regards to other hierarchical storage systems, such as NUMA or virtual memory, as well as for multiple levels of cache: once a sub-problem is small enough, it can be solved within a given level of the hierarchy, without accessing the higher (slower) levels.

Roundoff control

In computations with rounded arithmetic, e.g. with floating point numbers, a divide-and-conquer algorithm may yield more accurate results than a superficially equivalent iterative method. For example, one can add N numbers either by a simple loop that adds each datum to a single variable, or by a D&C algorithm called pairwise summation that breaks the data set into two halves, recursively computes the sum of each half, and then adds the two sums. While the second method performs the same number of additions as the first, and pays the overhead of the recursive calls, it is usually more accurate.[9]

Implementation issues

Recursion

Divide-and-conquer algorithms are naturally implemented as recursive procedures. In that case, the partial sub-problems leading to the one currently being solved are automatically stored in the procedure call stack.

Explicit stack

Divide and conquer algorithms can also be implemented by a non-recursive program that stores the partial sub-problems in some explicit data structure, such as a stack, queue, or priority queue. This approach allows more freedom in the choice of the sub-problem that is to be solved next, a feature that is important in some applications — e.g. in breadth-first recursion and the branch and bound method for function optimization. This approach is also the standard solution in programming languages that do not provide support for recursive procedures.

Stack size

In recursive implementations of D&C algorithms, one must make sure that there is sufficient memory allocated for the recursion stack, otherwise the execution may fail because of stack overflow. Fortunately, D&C algorithms that are time-efficient often have relatively small recursion depth. For example, the quicksort algorithm can be implemented so that it never requires more than $\log_2 n$ nested recursive calls to sort n items.

Stack overflow may be difficult to avoid when using recursive procedures, since many compilers assume that the recursion stack is a contiguous area of memory, and some allocate a fixed amount of space for it. Compilers may also save more information in the recursion stack than is strictly necessary, such as return address, unchanging parameters, and the internal variables of the procedure. Thus, the risk of stack overflow can be reduced by minimizing the parameters and internal variables of the recursive procedure, and/or by using an explicit stack structure.

Choosing the base cases

In any recursive algorithm, there is considerable freedom in the choice of the *base cases*, the small subproblems that are solved directly in order to terminate the recursion.

Choosing the smallest or simplest possible base cases is more elegant and usually leads to simpler programs, because there are fewer cases to consider and they are easier to solve. For example, an FFT algorithm could stop the recursion when the input is a single sample, and the quicksort list-sorting algorithm could stop when the input is the empty list; in both examples there is only one base case to consider, and it requires no processing.

On the other hand, efficiency often improves if the recursion is stopped at relatively large base cases, and these are solved non-recursively. This strategy avoids the overhead of recursive calls that do little or no work, and may also allow the use of specialized non-recursive algorithms that, for those base cases, are more efficient than explicit recursion. Since a D&C algorithm eventually reduces each problem or sub-problem instance to a large number of base instances, these often dominate the overall cost of the algorithm, especially when the splitting/joining overhead is low. Note that these considerations do not depend on whether recursion is implemented by the compiler or by an explicit stack.

Thus, for example, many library implementations of quicksort will switch to a simple loop-based insertion sort (or similar) algorithm once the number of items to be sorted is sufficiently small. Note that, if the empty list were the only base case, sorting a list with *n* entries would entail *n*+1 quicksort calls that would do nothing but return immediately. Increasing the base cases to lists of size 2 or less will eliminate most of those do-nothing calls, and more generally a base case larger than 2 is typically used to reduce the fraction of time spent in function-call overhead or stack manipulation.

Alternatively, one can employ large base cases that still use a divide-and-conquer algorithm, but implement the algorithm for predetermined set of fixed sizes where the algorithm can be completely unrolled into code that has no recursion, loops, or conditionals (related to the technique of partial evaluation). For example, this approach is used in some efficient FFT implementations, where the base cases are unrolled implementations of divide-and-conquer FFT algorithms for a set of fixed sizes.[10] The large number of separate base cases desirable to implement this strategy efficiently are sometimes produced by source code generation methods.[10]

The generalized version of this idea is known as recursion unrolling and various techniques have been proposed for automating the procedure of enlarging the base case.[11]

Sharing repeated subproblems

For some problems, the branched recursion may end up evaluating the same sub-problem many times over. In such cases it may be worth identifying and saving the solutions to these overlapping subproblems, a technique commonly known as memoization. Followed to the limit, it leads to bottom-up divide-and-conquer algorithms such as dynamic programming and chart parsing.

See also

- Mathematical induction
- The Master theorem
- The Akra-Bazzi method
- Binary search algorithm
- Quicksort
- Prune and search

References

[1] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, *Introduction to Algorithms* (MIT Press, 2000).

[2] Brassard, G. and Bratley, P. Fundamental of Algorithmics, Prentice-Hall, 1996.

[3] Anany V. Levitin, *Introduction to the Design and Analysis of Algorithms* (Addison Wesley, 2002).

[4] Donald E. Knuth, *The Art of Computer Programming: Volume 3, Sorting and Searching*, second edition (Addison-Wesley, 1998).

[5] Heideman, M. T., D. H. Johnson, and C. S. Burrus, "Gauss and the history of the fast Fourier transform," IEEE ASSP Magazine, 1, (4), 14–21 (1984)

[6] Knuth, Donald (1998). *The Art of Computer Programming: Volume 3 Sorting and Searching*. pp. 159. ISBN 0-201-89685-0.

[7] Karatsuba, Anatolii A.; Yuri P. Ofman (1962). "Умножение многозначных чисел на автоматах". *Doklady Akademii Nauk SSSR* **146**: 293–294. Translated in *Physics-Doklady* **7**: 595–596. 1963.

[8] M. Frigo; C. E. Leiserson, H. Prokop (1999). "Cache-oblivious algorithms". *Proc. 40th Symp. On the Foundations of Computer Science*.

[9] Nicholas J. Higham, "The accuracy of floating point summation", *SIAM J. Scientific Computing* **14** (4), 783–799 (1993).

[10] Frigo, M.; Johnson, S. G. (February 2005). "The design and implementation of FFTW3" (http://www.fftw.org/fftw-paper-ieee.pdf). *Proceedings of the IEEE* **93** (2): 216–231. doi:10.1109/JPROC.2004.840301. .

[11] Radu Rugina and Martin Rinard, " Recursion unrolling for divide and conquer programs (http://people.csail.mit.edu/rinard/paper/lcpc00.pdf)," in *Languages and Compilers for Parallel Computing*, chapter 3, pp. 34–48. *Lecture Notes in Computer Science* vol. 2017 (Berlin: Springer, 2001).

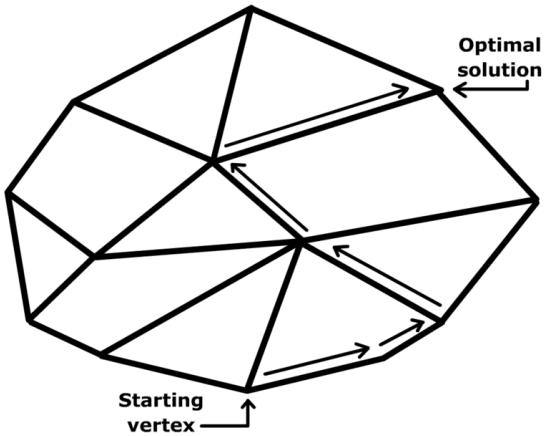
Simplex algorithm

In mathematical optimization theory, the **simplex algorithm**, created by the American mathematician George Dantzig in 1947, is a popular algorithm for numerically solving linear programming problems. The journal *Computing in Science and Engineering* listed it as one of the top 10 algorithms of the century.[1]

The method uses the concept of a simplex, which is a polytope of $N + 1$ vertices in N dimensions: a line segment in one dimension, a triangle in two dimensions, a tetrahedron in three-dimensional space and so forth.

Overview

Consider a linear programming problem,



A system of linear inequalities defines a polytope as a feasible region. The simplex algorithm begins at a starting vertex and moves along the edges of the polytope until it reaches the vertex of the optimum solution.

maximize $\mathbf{c}^T\mathbf{x}$

subject to $\mathbf{A}\mathbf{x} \leq \mathbf{b},\ \mathbf{x} \geq 0$

with $x = (x_1, \ldots, x_n)$ the variables of the problem, $c = (c_1, \ldots, c_n)$ a vector representing the linear form to optimize, A a rectangular p,n matrix and $b = (b_1, \ldots, b_p)$ the linear constraints.

In geometric terms, each inequality specifies a half-space in n-dimensional Euclidean space, and their intersection is the set of all feasible values the variables can take. The region is convex and either empty, unbounded, or a polytope.

The set of points where the objective function obtains a given value v is defined by the hyperplane $c^T x = v$. We are looking for the largest v such that the hyperplane still intersects the feasible region. As v increases, the hyperplane translates in the direction of the vector c. Intuitively, and indeed it can be shown by convexity, the last hyperplane to intersect the feasible region will either just graze a vertex of the polytope, or a whole edge or face. In the latter two cases, it is still the case that the endpoints of the edge or face will achieve the optimum value. Thus, the optimum value will always be achieved on (at least) one of the vertices of the polytope.

The simplex algorithm applies this insight by walking along edges of the (possibly unbounded) polytope to vertices with higher objective function value. When a local maximum is reached, by convexity it is also the global maximum and the algorithm terminates. It also finishes when an unbounded edge is visited, concluding that the problem has no solution. The algorithm always terminates because the number of vertices in the polytope is finite; moreover since we jump between vertices always in the same direction (that of the objective function), we hope that the number of vertices visited will be small. Usually there are more than one adjacent vertices which improve the objective function, so a *pivot rule* must be specified to determine which vertex to pick. The selection of this rule has a great

impact on the runtime of the algorithm.

In 1972, Klee and Minty[2] gave an example of a linear programming problem in which the polytope *P* is a distortion of an *n*-dimensional cube. They showed that the simplex method as formulated by Dantzig visits all 2^n vertices before arriving at the optimal vertex. This shows that the worst-case complexity of the algorithm is exponential time. Since then it has been shown that for almost every deterministic rule there is a family of simplices on which it performs badly. It is an open question if there is a pivot rule with polynomial time, or even sub-exponential worst-case complexity.

Nevertheless, the simplex method is remarkably efficient in practice. It has been known since the 1970s that it has polynomial-time average-case complexity under various distributions. These results on "random" matrices still didn't quite capture the desired intuition that the method works well on "typical" matrices. In 2001 Spielman and Teng introduced the notion of smoothed complexity to provide a more realistic analysis of the performance of algorithms.[3]

Other algorithms for solving linear programming problems are described in the linear programming article.

Algorithm

The simplex algorithm requires the linear programming problem to be in augmented form, so that the inequalities are replaced by equalities. The problem can then be written as follows in matrix form:

Maximize *Z* in:

$$\begin{bmatrix} 1 & -\mathbf{c}^T & 0 \\ 0 & \mathbf{A} & \mathbf{I} \end{bmatrix} \begin{bmatrix} Z \\ \mathbf{x} \\ \mathbf{x}_s \end{bmatrix} = \begin{bmatrix} 0 \\ \mathbf{b} \end{bmatrix}$$

$$\mathbf{x},\ \mathbf{x}_s \geq 0$$

where $x_s = (x_{s,1}; \ldots; x_{s,p})$ are the introduced slack variables from the augmentation process, ie the non-negative distances between the point and the hyperplanes representing the linear constraints *A*.

By definition, the vertices of the feasible region are each at the intersection of *n* hyperplanes (either from *A* or $\mathbf{x} \geq 0$). This corresponds to *n* zeros in the *n*+*p* variables of ($\mathbf{x}$, $\mathbf{x_s}$). Thus 2 feasible vertices are adjacent when they share *n*-1 zeros in the variables of ($\mathbf{x}$, $\mathbf{x_s}$). This is the interest of the augmented form notations : vertices and jumps along edges of the polytope are easy to write.

The simplex algorithm goes as follows :

- Start off by finding a feasible vertex. It will have at least *n* zeros in the variables of ($\mathbf{x}$, $\mathbf{x_s}$), that will be called the *n* current **non-basic variables**.
- Write *Z* as an affine function of the *n* current non-basic variables. This is done by transvections to move the non-zero coefficients in the first line of the matrix above.
- Now we want to jump to an adjacent feasible vertex to increase *Z*'s value. That means increasing the value of one of the non-basic variables, while keeping all *n*-1 others to zero. Among the *n* candidates in the adjacent feasible vertices, we choose that of greatest positive increase rate in *Z*, also called **direction of highest gradient**.
  - The changing non-basic variable is therefore easily identified as the one with maximum positive coefficient in *Z*.
  - If there are already no more positive coefficients in *Z* affine expression, then the solution vertex has been found and the algorithm terminates.
- Next we need to compute the coordinates of the new vertex we jump to. That vertex will have one of the *p* current basic variables set to 0, that variable must be found among the *p* candidates. By convexity of the feasible polytope, the correct basic variable is the one that, when set to 0, **minimizes the change in the moving non-basic variable**. This is easily found by computing *p* ratios of coefficients and taking the lowest ratio. That new variable replaces the moving one in the *n* non-basic variables and the algorithm loops back to writing *Z* as a function of

these.

Example

Rewrite $\mathbf{x} = (x, y, z)$ and $\mathbf{x_s} = (s, t)$:

$$\begin{bmatrix} 1 & -2 & -3 & -4 & 0 & 0 \\ 0 & 3 & 2 & 1 & 1 & 0 \\ 0 & 2 & 5 & 3 & 0 & 1 \end{bmatrix} \begin{bmatrix} Z \\ x \\ y \\ z \\ s \\ t \end{bmatrix} = \begin{bmatrix} 0 \\ 10 \\ 15 \end{bmatrix}$$

x=0 is clearly a feasible vertex so we start off with it : the 3 current non-basic variables are *x*, *y* and *z*. Luckily *Z* is already expressed as an affine function of *x*,*y*,*z* so no transvections need to be done at this step. Here *Z* 's greatest coefficient is -4, so the direction with highest gradient is *z*. We then need to compute the coordinates of the vertex we jump to increasing *z*. That will result in setting either *s* or *t* to 0 and the correct one must be found. For *s*, the change in *z* equals 10/1=10 ; for *t* it is 15/3=5. *t* is the correct one because it minimizes that change. The new vertex is thus *x*=*y*=*t*=0. Rewrite *Z* as an affine function of these new non-basic variables :

$$\begin{bmatrix} 1 & \frac{2}{3} & \frac{11}{3} & 0 & 0 & \frac{4}{3} \\ 0 & 3 & 2 & 1 & 1 & 0 \\ 0 & 2 & 5 & 3 & 0 & 1 \end{bmatrix} \begin{bmatrix} Z \\ x \\ y \\ z \\ s \\ t \end{bmatrix} = \begin{bmatrix} 20 \\ 10 \\ 15 \end{bmatrix}$$

Now all coefficients on the first row of the matrix have become nonnegative. That means *Z* cannot be improved by increasing any of the current non-basic variables. The algorithm terminates and the solution is the vertex *x*=*y*=*t*=0. On that vertex *Z*=20 and this is its maximum value. Usually the coordinates of the solution vertex are needed in the standard variables *x*, *y*, *z* ; so a little substitution yields *x*=0, *y*=0 and *z*=5.

Implementation

The tableau form used above to describe the algorithm lends itself to an immediate implementation in which the tableau is maintained as a rectangular (m+1)-by-(m+n+1) array. It is straightforward to avoid storing the m explicit columns of the identity matrix that will occur within the tableau by virtue of **B** being a subset of the columns of $\begin{bmatrix} \mathbf{A} & \mathbf{I} \end{bmatrix}$. This implementation is referred to as the standard simplex method. The storage and computation overhead are such that the standard simplex method is a prohibitively expensive approach to solving large linear programming problems.

In each simplex iteration, the only data required are the first row of the tableau, the (pivotal) column of the tableau corresponding to the entering variable and the right-hand-side. The latter can be updated using the pivotal column and the first row of the tableau can be updated using the (pivotal) row corresponding to the leaving variable. Both the pivotal column and pivotal row may be computed directly using the solutions of linear systems of equations involving the matrix **B** and a matrix-vector product using **A**. These observations motivate the revised simplex method, for which implementations are distinguished by their invertible representation of **B**.

In large linear programming problems **A** is typically a sparse matrix and, when the resulting sparsity of **B** is exploited when maintaining its invertible representation, the revised simplex method is a vastly more efficient solution procedure than the standard simplex method. Commercial simplex solvers are based on the primal (or dual) revised simplex method.

See also

- Nelder-Mead method
- Fourier-Motzkin elimination
- Bland's anti-cycling rule
- Karmarkar's algorithm

Further reading

- Dmitris Alevras and Manfred W. Padberg, *Linear Optimization and Extensions: Problems and Extensions*, Universitext, Springer-Verlag, 2001. (Problems from Padberg with solutions.)
- J. E. Beasley, editor. *Advances in Linear and Integer Programing*. Oxford Science, 1996. (Collection of surveys)
- R.G. Bland, New finite pivoting rules for the simplex method, *Math. Oper. Res.* 2 (1977) 103–107.
- Karl-Heinz Borgwardt, *The Simplex Algorithm: A Probabilistic Analysis*, Algorithms and Combinatorics, Volume 1, Springer-Verlag, 1987. (Average behavior on random problems)
- Richard W. Cottle, ed. *The Basic George B. Dantzig*. Stanford Business Books, Stanford University Press, Stanford, California, 2003. (Selected papers by George B. Dantzig)
- George B. Dantzig and Mukund N. Thapa. 1997. *Linear programming 1: Introduction*. Springer-Verlag.
- George B. Dantzig and Mukund N. Thapa. 2003. *Linear Programming 2: Theory and Extensions*. Springer-Verlag.
- Komei Fukuda and Tamás Terlaky, Criss-cross methods: A fresh view on pivot algorithms. (1997) *Mathematical Programming* Series B, Vol 79, Nos. 1--3, 369--395. (Invited survey, from the International Symposium on Mathematical Programming.)
- Kattta G. Murty, *Linear Programming*, Wiley, 1983. (comprehensive textbook and reference, through ellipsiodal algorithm of Khachiyan)
- Evar D. Nering and Albert W. Tucker, 1993, *Linear Programs and Related Problems*, Academic Press. (elementary)
- M. Padberg, *Linear Optimization and Extensions*, Second Edition, Springer-Verlag, 1999. (carefully written account of primal and dual simplex algorithms and projective algorithms, with an introduction to integer linear programming --- featuring the traveling salesman problem for Odysseus.)
- Christos H. Papadimitriou and Kenneth Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Corrected republication with a new preface, Dover. (computer science)
- Alexander Schrijver, *Theory of Linear and Integer Programming*. John Wiley & sons, 1998, ISBN 0-471-98232-6 (mathematical)
- Michael J. Todd (February 2002). "The many facets of linear programming". *Mathematical Programming* **91** (3). (Invited survey, from the International Symposium on Mathematical Programming.)
- Greenberg, Harvey J., *Klee-Minty Polytope Shows Exponential Time Complexity of Simplex Method* University of Colorado at Denver (1997) PDF download [4]
- Frederick S. Hillier and Gerald J. Lieberman: *Introduction to Operations Research*, 8th edition. McGraw-Hill. ISBN 0-07-123828-X
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Section 29.3: The simplex algorithm, pp.790–804.
- Hamdy A. Taha: *Operations Research: An Introduction*, 8th ed., Prentice Hall, 2007. ISBN 0-13-188923-0
- Richard B. Darst: *Introduction to Linear Programming: Applications and Extensions*
- Press, William H., et. al.: *Numerical Recipes in C*, Second Edition, Cambridge University Press, ISBN 0-521-43108-5. Section 10.8: Linear Programming and the Simplex Method, pp.430-444. (A different tableau format, for which the use of the word pivot is quite intuitive. Computer code in C.)

External links

- An Introduction to Linear Programming and the Simplex Algorithm [5] by Spyros Reveliotis of the Georgia Institute of Technology.
- LP-Explorer [6] A Java-based tool which, for problems in two variables, relates the algebraic and geometric views of the tableau simplex method. Also illustrates the sensitivity of the solution to changes in the right-hand-side.
- Java-based interactive simplex tool [7] hosted by Argonne National Laboratory.
- Tutorial for The Simplex Method [8] by Stefan Waner, hofstra.edu. Interactive worked example.
- A simple example - step by step [9] by Mazoo's Learning Blog.
- simplex me - the simple simplex solver [10] (Solve your linear problems with simplex algorithm online - multilanguage, PDF Export)
- Simplex Method [11] A good tutorial for Simplex Method with examples (also two-phase and M-method).
- PHPSimplex [12] Other good tutorial for Simplex Method with the Two-Phase Simplex and Graphical methods, examples, and a tool to solve Simplex problems online.
- Simplex Method Tool [13] Quick-loading JavaScript-based web page that solves Simplex problems
- LiSimplex Solver [14] LiSimplex - The OpenSource Java Simplex Solver

References

[1] *Computing in Science and Engineering*, volume 2, no. 1, 2000

[2] Greenberg, cites: V. Klee and G.J. Minty. "How Good is the Simplex Algorithm?" In O. Shisha, editor, *Inequalities, III*, pages 159–175. Academic Press, New York, NY, 1972

[3] Spielman, Daniel; Teng, Shang-Hua (2001), "Smoothed analysis of algorithms: why the simplex algorithm usually takes polynomial time", *Proceedings of the Thirty-Third Annual ACM Symposium on Theory of Computing*, ACM, pp. 296–305, doi:10.1145/380752.380813, arXiv:cs/0111050, ISBN 978-1-58113-349-3

[4] http://glossary.computing.society.informs.org/notes/Klee-Minty.pdf

[5] http://www.isye.gatech.edu/~spyros/LP/LP.html

[6] http://www.maths.ed.ac.uk/LP-Explorer/

[7] http://www-fp.mcs.anl.gov/otc/Guide/CaseStudies/simplex/

[8] http://people.hofstra.edu/faculty/Stefan_Waner/RealWorld/tutorialsf4/frames4_3.html

[9] http://learning.mazoo.net/archives/001240.html

[10] http://www.simplexme.com

[11] http://www.math.cuhk.edu.hk/~wei/lpch3.pdf

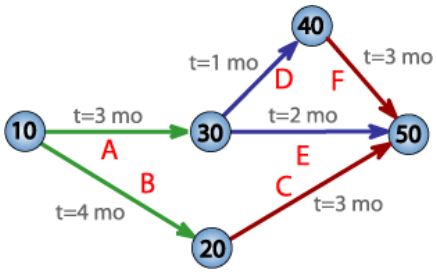
[12] http://www.phpsimplex.com/en/index.htm

[13] http://people.hofstra.edu/faculty/Stefan_Waner/RealWorld/simplex.html

[14] http://translate.google.com/translate?js=y&prev=_t&hl=pt-BR&ie=UTF-8&layout=1&eotf=1&u=http%3A%2F%2Fricardogobbo.wordpress.com%2F2009%2F07%2F17%2Flisimplex-simplex-solver-implementacao-de-um-solver-simplex-em-java%2F&sl=pt&tl=en

Critical path method

The **critical path method (CPM)** or **critical path analysis**, is a mathematically based algorithm for scheduling a set of project activities. It is an important tool for effective project management.



PERT chart for a project with five milestones (10 through 50) and six activities (A through F). The project has two critical paths: activities B and C, or A, D, and F – giving a minimum project time of 7 months with fast tracking. Activity E is sub-critical, and has a float of 2 months.

History

It was developed in the 1950s by the DuPont Corporation at about the same time that Booz Allen Hamilton and the US Navy were developing the Program Evaluation and Review Technique [1] Today, it is commonly used with all forms of projects, including construction, software development, research projects, product development, engineering, and plant maintenance, among others. Any project with interdependent activities can apply this method of scheduling.

Basic technique

The essential technique for using CPM [2] is to construct a model of the project that includes the following:

1. A list of all activities required to complete the project (typically categorized within a work breakdown structure),
2. The time (duration) that each activity will take to completion, and
3. The dependencies between the activities

Using these values, CPM calculates the longest path of planned activities to the end of the project, and the earliest and latest that each activity can start and finish without making the project longer. This process determines which activities are "critical" (i.e., on the longest path) and which have "total float" (i.e., can be delayed without making the project longer). In project management, a **critical path** is the sequence of project network activities which add up to the longest overall duration. This determines the shortest time possible to complete the project. Any delay of an activity on the critical path directly impacts the planned project completion date (i.e. there is no float on the critical path). A project can have several, parallel, near critical paths. An additional parallel path through the network with the total durations shorter than the critical path is called a sub-critical or non-critical path.

These results allow managers to prioritize activities for the effective management of project completion, and to shorten the planned critical path of a project by pruning critical path activities, by "**fast tracking**" (i.e., performing more activities in parallel), and/or by "**crashing the critical path**" (i.e., shortening the durations of critical path activities by adding resources).

Expansion

Originally, the critical path method considered only logical dependencies between terminal elements. Since then, it has been expanded to allow for the inclusion of resources related to each activity, through processes called activity-based resource assignments and resource leveling. A resource-leveled schedule may include delays due to resource bottlenecks (i.e., unavailability of a resource at the required time), and may cause a previously shorter path to become the longest or most "resource critical" path. A related concept is called the critical chain, which attempts to protect activity and project durations from unforeseen delays due to resource constraints.

Since project schedules change on a regular basis, CPM allows continuous monitoring of the schedule, allows the project manager to track the critical activities, and alerts the project manager to the possibility that non-critical activities may be delayed beyond their total float, thus creating a new critical path and delaying project completion. In addition, the method can easily incorporate the concepts of stochastic predictions, using the Program Evaluation

and Review Technique (PERT) and event chain methodology.

Currently, there are several software solutions available in industry that use the CPM method of scheduling, see list of project management software. However, the method was developed and used without the aid of computers.

Flexibility

A schedule generated using critical path techniques often is not realised precisely, as estimations are used to calculate times: if one mistake is made, the results of the analysis may change. This could cause an upset in the implementation of a project if the estimates are blindly believed, and if changes are not addressed promptly. However, the structure of critical path analysis is such that the variance from the original schedule caused by any change can be measured, and its impact either ameliorated or adjusted for. Indeed, an important element of project postmortem analysis is the As Built Critical Path (ABCP), which analyzes the specific causes and impacts of changes between the planned schedule and eventual schedule as actually implemented.

Running time

Given a graph G=G(N,E) of N nodes and E edges, if we use Big O notation, the CPM algorithm takes O(E) to complete, since topological ordering of a graph takes O(E) and every edge is considered only twice, which means linear time in number of edges.

See also

- Gantt chart
- List of project management software
- List of project management topics
- Program Evaluation and Review Technique (PERT)
- Project
- Project management
- Project planning
- Shifting bottleneck heuristic
- Work breakdown structure
- Backward induction

Further reading

- Project Management Institute (2003). *A Guide To The Project Management Body Of Knowledge* (3rd ed.). Project Management Institute. ISBN 1-930699-45-X.
- Klastorin, Ted (2003). *Project Management: Tools and Trade-offs* (3rd ed.). Wiley. ISBN 978-0471413844.
- Heerkens, Gary (2001). *Project Management (The Briefcase Book Series)*. McGraw–Hill. ISBN 0-07-137952-5.
- Kerzner, Harold (2003). *Project Management: A Systems Approach to Planning, Scheduling, and Controlling* (8th ed.). ISBN 0-471-22577-0.
- Lewis, James (2002). *Fundamentals of Project Management* (2nd ed.). American Management Association. ISBN 0-8144-7132-3.
- Milosevic, Dragan Z. (2003). *Project Management ToolBox: Tools and Techniques for the Practicing Project Manager*. Wiley. ISBN 978-0471208228.
- Woolf, Murray B. (2007). *Faster Construction Projects with CPM Scheduling*. McGraw Hill. ISBN 978-0071486606.

External links

- Critical path web calculator [3]
- A Few Critical Path Articles [4]
- A good slide show explaining critical path concepts [5]
- Critical Path Java Applet [6]

References

[1] Newell, M; Grashina, M (2003). *The Project Management Question and Answer Book*. American Management Association. p. 98.
[2] Samuel L. Baker, Ph.D. "Critical Path Method (CPM)" (http://hspm.sph.sc.edu/COURSES/J716/CPM/CPM.html) *University of South Carolina*, Health Services Policy and Management Courses
[3] http://sporkforge.com/sched/critical_path.php
[4] http://www.pmhut.com/category/time-management/critical-path/
[5] http://www.slideshare.net/dmdk12/the-network-diagram-and-critical-path
[6] http://www.cut-the-knot.org/Curriculum/Combinatorics/CriticalPath.shtml

Monte Carlo method

Monte Carlo methods (or **Monte Carlo experiments**) are a class of computational algorithms that rely on repeated random sampling to compute their results. Monte Carlo methods are often used in simulating physical and mathematical systems. Because of their reliance on repeated computation of random or pseudo-random numbers, these methods are most suited to calculation by a computer and tend to be used when it is unfeasible or impossible to compute an exact result with a deterministic algorithm.[1]

Monte Carlo simulation methods are especially useful in studying systems with a large number of coupled degrees of freedom, such as fluids, disordered materials, strongly coupled solids, and cellular structures (see cellular Potts model). More broadly, Monte Carlo methods are useful for modeling phenomena with significant uncertainty in inputs, such as the calculation of risk in business. These methods are also widely used in mathematics: a classic use is for the evaluation of definite integrals, particularly multidimensional integrals with complicated boundary conditions. It is a widely successful method in risk analysis when compared with alternative methods or human intuition. When Monte Carlo simulations have been applied in space exploration and oil exploration, actual observations of failures, cost overruns and schedule overruns are routinely better predicted by the simulations than by human intuition or alternative "soft" methods.[2]

The term "Monte Carlo method" was coined in the 1940s by physicists working on nuclear weapon projects in the Los Alamos National Laboratory.[3]

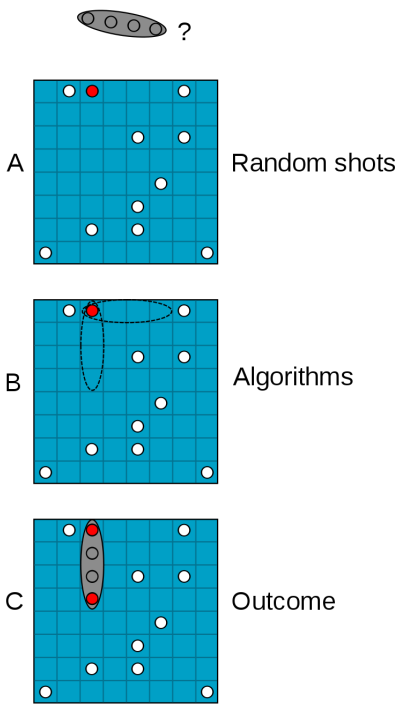
Overview

There is no single Monte Carlo method; instead, the term describes a large and widely-used class of approaches. However, these approaches tend to follow a particular pattern:

1. Define a domain of possible inputs.
2. Generate inputs randomly from the domain using a certain specified probability distribution.
3. Perform a deterministic computation using the inputs.
4. Aggregate the results of the individual computations into the final result.

For example, the value of π can be approximated using a Monte Carlo method:

1. Draw a square on the ground, then inscribe a circle within it. From plane geometry, the ratio of the area of an inscribed circle to that of the surrounding square is $\pi/4$.
2. Uniformly scatter some objects of uniform size throughout the square. For example, grains of rice or sand.
3. Since the two areas are in the ratio $\pi/4$, the objects should fall in the areas in approximately the same ratio. Thus, counting the number of objects in the circle and dividing by the total number of objects in the square will yield an approximation for $\pi/4$.
4. Multiplying the result by 4 will then yield an approximation for π itself.



The Monte Carlo method can be illustrated as a game of Battleship. First a player makes some random shots. Next the player applies algorithms (i.e. a battleship is four dots in the vertical or horizontal direction). Finally based on the outcome of the random sampling and the algorithm the player can determine the likely locations of the other player's ships.

Notice how the π approximation follows the general pattern of Monte Carlo algorithms. First, we define a domain of inputs: in this case, it's the square which circumscribes our circle. Next, we generate inputs randomly (scatter individual grains within the square), then perform a computation on each input (test whether it falls within the circle). At the end, we aggregate the results into our final result, the approximation of π . Note, also, two other common properties of Monte Carlo methods: the computation's reliance on good random numbers, and its slow convergence to a better approximation as more data points are sampled. If grains are purposefully dropped into only, for example, the center of the circle, they will not be uniformly distributed, and so our approximation will be poor. An approximation will also be poor if only a few grains are randomly dropped into the whole square. Thus, the approximation of π will become more accurate both as the grains are dropped more uniformly and as more are dropped.

History

Enrico Fermi in the 1930s and Stanisław Ulam in 1946 first had the idea. Ulam later contacted John von Neumann to work on it.[4]

Physicists at Los Alamos Scientific Laboratory were investigating radiation shielding and the distance that neutrons would likely travel through various materials. Despite having most of the necessary data, such as the average distance a neutron would travel in a substance before it collided with an atomic nucleus or how much energy the neutron was likely to give off following a collision, the problem could not be solved with analytical calculations. John von Neumann and Stanislaw Ulam suggested that the problem be solved by modeling the experiment on a computer using chance. Being secret, their work required a code name. Von Neumann chose the name "Monte

Carlo". The name is a reference to the Monte Carlo Casino in Monaco where Ulam's uncle would borrow money to gamble.[1] [5] [6]

Random methods of computation and experimentation (generally considered forms of stochastic simulation) can be arguably traced back to the earliest pioneers of probability theory (see, e.g., Buffon's needle, and the work on small samples by William Sealy Gosset), but are more specifically traced to the pre-electronic computing era. The general difference usually described about a Monte Carlo form of simulation is that it systematically "inverts" the typical mode of simulation, treating deterministic problems by *first* finding a probabilistic analog (see Simulated annealing). Previous methods of simulation and statistical sampling generally did the opposite: using simulation to test a previously understood deterministic problem. Though examples of an "inverted" approach do exist historically, they were not considered a general method until the popularity of the Monte Carlo method spread.

Monte Carlo methods were central to the simulations required for the Manhattan Project, though were severely limited by the computational tools at the time. Therefore, it was only after electronic computers were first built (from 1945 on) that Monte Carlo methods began to be studied in depth. In the 1950s they were used at Los Alamos for early work relating to the development of the hydrogen bomb, and became popularized in the fields of physics, physical chemistry, and operations research. The Rand Corporation and the U.S. Air Force were two of the major organizations responsible for funding and disseminating information on Monte Carlo methods during this time, and they began to find a wide application in many different fields.

Uses of Monte Carlo methods require large amounts of random numbers, and it was their use that spurred the development of pseudorandom number generators, which were far quicker to use than the tables of random numbers which had been previously used for statistical sampling.

Applications

As mentioned, Monte Carlo simulation methods are especially useful for modeling phenomena with significant uncertainty in inputs and in studying systems with a large number of coupled degrees of freedom. Specific areas of application include:

Physical sciences

Monte Carlo methods are very important in computational physics, physical chemistry, and related applied fields, and have diverse applications from complicated quantum chromodynamics calculations to designing heat shields and aerodynamic forms. The Monte Carlo method is widely used in statistical physics, particularly Monte Carlo molecular modeling as an alternative for computational molecular dynamics as well as to compute statistical field theories of simple particle and polymer models [7] ; see Monte Carlo method in statistical physics. In experimental particle physics, these methods are used for designing detectors, understanding their behavior and comparing experimental data to theory, or on vastly large scale of the galaxy modelling.[8]

Monte Carlo methods are also used in the ensemble models that form the basis of modern weather forecasting operations.

Design and visuals

Monte Carlo methods have also proven efficient in solving coupled integral differential equations of radiation fields and energy transport, and thus these methods have been used in global illumination computations which produce photorealistic images of virtual 3D models, with applications in video games, architecture, design, computer generated films, and cinematic special effects.

Finance and business

Monte Carlo methods in finance are often used to calculate the value of companies, to evaluate investments in projects at a business unit or corporate level, or to evaluate financial derivatives. Monte Carlo methods used in these cases allow the construction of stochastic or probabilistic financial models as opposed to the traditional static and deterministic models, thereby enhancing the treatment of uncertainty in the calculation. For use in the insurance industry, see stochastic modelling.

Telecommunications

When planning a wireless network, design must be proved to work for a wide variety of scenarios that depend mainly on the number of users, their locations and the services they want to use. Monte Carlo methods are typically used to generate these users and their states. The network performance is then evaluated and, if results are not satisfactory, the network design goes through an optimization process.

Games

Monte Carlo methods have recently been applied in game playing related artificial intelligence theory. Most notably the game of Go and Battleship and have seen remarkably successful Monte Carlo algorithm based computer players. One of the main problems that this approach has in game playing is that it sometimes misses an isolated, very good move. These approaches are often strong strategically but weak tactically, as tactical decisions tend to rely on a small number of crucial moves which are easily missed by the randomly searching Monte Carlo algorithm.

Monte Carlo simulation versus “what if” scenarios

The opposite of Monte Carlo simulation might be considered deterministic modeling using single-point estimates. Each uncertain variable within a model is assigned a “best guess” estimate. Various combinations of each input variable are manually chosen (such as best case, worst case, and most likely case), and the results recorded for each so-called “what if” scenario.[9]

By contrast, Monte Carlo simulation considers random sampling of probability distribution functions as model inputs to produce hundreds or thousands of possible outcomes instead of a few discrete scenarios. The results provide probabilities of different outcomes occurring.[10]

For example, a comparison of a spreadsheet cost construction model run using traditional “what if” scenarios, and then run again with Monte Carlo simulation and Triangular probability distributions shows that the Monte Carlo analysis has a narrower range than the “what if” analysis. This is because the “what if” analysis gives equal weight to all scenarios.[11]

For further discussion, see quantifying uncertainty under corporate finance.

Use in mathematics

In general, Monte Carlo methods are used in mathematics to solve various problems by generating suitable random numbers and observing that fraction of the numbers which obeys some property or properties. The method is useful for obtaining numerical solutions to problems which are too complicated to solve analytically. The most common application of the Monte Carlo method is Monte Carlo integration.

Integration

Deterministic methods of numerical integration usually operate by taking a number of evenly spaced samples from a function. In general, this works very well for functions of one variable. However, for functions of vectors, deterministic quadrature methods can be very inefficient. To numerically integrate a function of a two-dimensional vector, equally spaced grid points over a two-dimensional surface are required. For instance a 10x10 grid requires 100 points. If the vector has 100 dimensions, the same spacing on the grid would require 10^{100} points—far too many to be computed. 100 dimensions is by no means unusual, since in many physical problems, a "dimension" is equivalent to a degree of freedom. (See Curse of dimensionality.)

Monte Carlo methods provide a way out of this exponential time-increase. As long as the function in question is reasonably well-behaved, it can be estimated by randomly selecting points in 100-dimensional space, and taking some kind of average of the function values at these points. By the law of large numbers, this method will display $1/\sqrt{N}$ convergence—i.e. quadrupling the number of sampled points will halve the error, regardless of the number of dimensions.

A refinement of this method is to somehow make the points random, but more likely to come from regions of high contribution to the integral than from regions of low contribution. In other words, the points should be drawn from a distribution similar in form to the integrand. Understandably, doing this precisely is just as difficult as solving the integral in the first place, but there are approximate methods available: from simply making up an integrable function thought to be similar, to one of the adaptive routines discussed in the topics listed below.

A similar approach involves using low-discrepancy sequences instead—the quasi-Monte Carlo method. Quasi-Monte Carlo methods can often be more efficient at numerical integration because the sequence "fills" the area better in a sense and samples more of the most important points that can make the simulation converge to the desired solution more quickly.

Integration methods

- Direct sampling methods
  - Importance sampling
  - Stratified sampling
  - Recursive stratified sampling
  - VEGAS algorithm
- Random walk Monte Carlo including Markov chains
  - Metropolis-Hastings algorithm
- Gibbs sampling

Optimization

Most Monte Carlo optimization methods are based on random walks. Essentially, the program will move around a marker in multi-dimensional space, tending to move in directions which lead to a lower function, but sometimes moving against the gradient.

Another powerful and very popular application for random numbers in numerical simulation is in numerical optimization. These problems use functions of some often large-dimensional vector that are to be minimized (or maximized). Many problems can be phrased in this way: for example a computer chess program could be seen as trying to find the optimal set of, say, 10 moves which produces the best evaluation function at the end. The traveling salesman problem is another optimization problem. There are also applications to engineering design, such as multidisciplinary design optimization.

Optimization methods

- Evolution strategy
- Genetic algorithms
- Parallel tempering
- Simulated annealing
- Stochastic optimization
- Stochastic tunneling

Inverse problems

Probabilistic formulation of inverse problems leads to the definition of a probability distribution in the model space. This probability distribution combines *a priori* information with new information obtained by measuring some observable parameters (data). As, in the general case, the theory linking data with model parameters is nonlinear, the a posteriori probability in the model space may not be easy to describe (it may be multimodal, some moments may not be defined, etc.).

When analyzing an inverse problem, obtaining a maximum likelihood model is usually not sufficient, as we normally also wish to have information on the resolution power of the data. In the general case we may have a large number of model parameters, and an inspection of the marginal probability densities of interest may be impractical, or even useless. But it is possible to pseudorandomly generate a large collection of models according to the posterior probability distribution and to analyze and display the models in such a way that information on the relative likelihoods of model properties is conveyed to the spectator. This can be accomplished by means of an efficient Monte Carlo method, even in cases where no explicit formula for the a priori distribution is available.

The best-known importance sampling method, the Metropolis algorithm, can be generalized, and this gives a method that allows analysis of (possibly highly nonlinear) inverse problems with complex a priori information and data with an arbitrary noise distribution. For details, see Mosegaard and Tarantola (1995),[12] or Tarantola (2005).[13]

Computational mathematics

Monte Carlo methods are useful in many areas of computational mathematics, where a *lucky choice* can find the correct result. A classic example is Rabin's algorithm for primality testing: for any n which is not prime, a random x has at least a 75% chance of proving that n is not prime. Hence, if n is not prime, but x says that it might be, we have observed at most a 1-in-4 event. If 10 different random x say that "n is probably prime" when it is not, we have observed a one-in-a-million event. In general a Monte Carlo algorithm of this kind produces one correct answer with a guarantee ***n* is composite, and *x* proves it so**, but another one without, but with a guarantee of not getting this answer when it is wrong **too often —** in this case at most 25% of the time. See also Las Vegas algorithm for a related, but different, idea.

Monte Carlo and random numbers

Interestingly, Monte Carlo simulation methods do not always require truly random numbers to be useful — while for some applications, such as primality testing, unpredictability is vital (see Davenport (1995)).[14] Many of the most useful techniques use deterministic, pseudo-random sequences, making it easy to test and re-run simulations. The only quality usually necessary to make good simulations is for the pseudo-random sequence to appear "random enough" in a certain sense.

What this means depends on the application, but typically they should pass a series of statistical tests. Testing that the numbers are uniformly distributed or follow another desired distribution when a large enough number of elements of the sequence are considered is one of the simplest, and most common ones.

See also

General

- Auxiliary field Monte Carlo
- Bootstrapping (statistics)
- Demon algorithm
- Evolutionary computation
- FERMIAC
- Markov chain
- Molecular dynamics
- Monte Carlo option model
- Monte Carlo integration
- Quasi-Monte Carlo method
- Random number generator
- Randomness
- Resampling (statistics)

Application areas

- Graphics, particularly for ray tracing; a version of the Metropolis-Hastings algorithm is also used for ray tracing where it is known as Metropolis light transport
- Modeling light transport in biological tissue
- Monte Carlo methods in finance
- Reliability engineering
- In simulated annealing for protein structure prediction
- In semiconductor device research, to model the transport of current carriers
- Environmental science, dealing with contaminant behavior
- Search And Rescue and Counter-Pollution. Models used to predict the drift of a life raft or movement of an oil slick at sea.
- In probabilistic design for simulating and understanding the effects of variability
- In physical chemistry, particularly for simulations involving atomic clusters
- In biomolecular simulations
- In polymer physics
  - Bond fluctuation model
- In computer science
  - Monte Carlo algorithm
  - Las Vegas algorithm

  - LURCH
  - Computer go
  - General Game Playing
- Modeling the movement of impurity atoms (or ions) in plasmas in existing and tokamaks (e.g.: DIVIMP).
- Nuclear and particle physics codes using the Monte Carlo method:
  - GEANT — CERN's simulation of high energy particles interacting with a detector.
  - FLUKA — INFN and CERN's simulation package for the interaction and transport of particles and nuclei in matter
  - SRIM, a code to calculate the penetration and energy deposition of ions in matter.
  - CompHEP, PYTHIA — Monte-Carlo generators of particle collisions
  - MCNP(X) - LANL's radiation transport codes
  - MCU: universal computer code for simulation of particle transport (neutrons, photons, electrons) in three-dimensional systems by means of the Monte Carlo method
  - EGS — Stanford's simulation code for coupled transport of electrons and photons
  - PEREGRINE: LLNL's Monte Carlo tool for radiation therapy dose calculations
  - BEAMnrc — Monte Carlo code system for modeling radiotherapy sources (LINAC's)
  - PENELOPE — Monte Carlo for coupled transport of photons and electrons, with applications in radiotherapy
  - MONK — Serco Assurance's code for the calculation of k-effective of nuclear systems
- Modelling of foam and cellular structures
- Modeling of tissue morphogenesis
- Computation of holograms
- Phylogenetic analysis, i.e. Bayesian inference, Markov chain Monte Carlo

Other methods employing Monte Carlo

- Assorted random models, e.g. self-organized criticality
- Direct simulation Monte Carlo
- Dynamic Monte Carlo method
- Kinetic Monte Carlo
- Quantum Monte Carlo
- Quasi-Monte Carlo method using low-discrepancy sequences and self avoiding walks
- Semiconductor charge transport and the like
- Electron microscopy beam-sample interactions
- Stochastic optimization
- Cellular Potts model
- Markov chain Monte Carlo
- Cross-entropy method
- Applied information economics
- Monte Carlo localization
- Evidence-based Scheduling
- Binary collision approximation

References

- Metropolis, N.; Ulam, S. (1949). "The Monte Carlo Method" [15]. *Journal of the American Statistical Association* (American Statistical Association) **44** (247): 335–341. doi:10.2307/2280232. PMID 18139350.
- Metropolis, Nicholas; Rosenbluth, Arianna W.; Rosenbluth, Marshall N.; Teller, Augusta H.; Teller, Edward (1953). "Equation of State Calculations by Fast Computing Machines". *Journal of Chemical Physics* **21** (6): 1087. doi:10.1063/1.1699114.
- Hammersley, J. M.; Handscomb, D. C. (1975). *Monte Carlo Methods*. London: Methuen. ISBN 0416523404.
- Kahneman, D.; Tversky, A. (1982). *Judgement under Uncertainty: Heuristics and Biases*. Cambridge University Press.
- Gould, Harvey; Tobochnik, Jan (1988). *An Introduction to Computer Simulation Methods, Part 2, Applications to Physical Systems*. Reading: Addison-Wesley. ISBN 020116504X.
- Binder, Kurt (1995). *The Monte Carlo Method in Condensed Matter Physics*. New York: Springer. ISBN 0387543694.
- Berg, Bernd A. (2004). *Markov Chain Monte Carlo Simulations and Their Statistical Analysis (With Web-Based Fortran Code)*. Hackensack, NJ: World Scientific. ISBN 9812389350.
- Caflisch, R. E. (1998). *Monte Carlo and quasi-Monte Carlo methods*. Acta Numerica. **7**. Cambridge University Press. pp. 1–49.
- Doucet, Arnaud; Freitas, Nando de; Gordon, Neil (2001). *Sequential Monte Carlo methods in practice*. New York: Springer. ISBN 0387951466.
- Fishman, G. S. (1995). *Monte Carlo: Concepts, Algorithms, and Applications*. New York: Springer. ISBN 038794527X.
- MacKeown, P. Kevin (1997). *Stochastic Simulation in Physics*. New York: Springer. ISBN 9813083263.
- Robert, C. P.; Casella, G. (2004). *Monte Carlo Statistical Methods* (2nd ed.). New York: Springer. ISBN 0387212396.
- Rubinstein, R. Y.; Kroese, D. P. (2007). *Simulation and the Monte Carlo Method* (2nd ed.). New York: John Wiley & Sons. ISBN 9780470177938.
- Mosegaard, Klaus; Tarantola, Albert (1995). "Monte Carlo sampling of solutions to inverse problems". *J. Geophys. Res.* **100** (B7): 12431–12447. doi:10.1029/94JB03097.
- Tarantola, Albert (2005). *Inverse Problem Theory* [16]. Philadelphia: Society for Industrial and Applied Mathematics. ISBN 0898715725.

External links

- Overview and reference list [17], Mathworld
- Introduction to Monte Carlo Methods [18], Computational Science Education Project
- Overview of formulas used in Monte Carlo simulation [19], the Quant Equation Archive, at sitmo.com
- The Basics of Monte Carlo Simulations [20], University of Nebraska-Lincoln
- Introduction to Monte Carlo simulation [21] (for Excel), Wayne L. Winston
- Monte Carlo Methods - Overview and Concept [22], brighton-webs.co.uk
- Molecular Monte Carlo Intro [23], Cooper Union
- Monte Carlo techniques applied in physics [24]
- MonteCarlo Simulation in Finance [25], global-derivatives.com
- Approximation of π with the Monte Carlo Method [26]
- Risk Analysis in Investment Appraisal [27], The Application of Monte Carlo Methodology in Project Appraisal, Savvakis C. Savvides
- Probabilistic Assessment of Structures using the Monte Carlo method [28], Wikiuniversity paper for students of Structural Engineering

- A very intuitive and comprehensive introduction to Quasi-Monte Carlo methods [29]
- Pricing using Monte Carlo simulation [30], a practical example, Prof. Giancarlo Vercellino

Software

- The BUGS project [31] (including WinBUGS and OpenBUGS)
- Monte Carlo Simulation, Resampling, Bootstrap tool [32]
- YASAI: Yet Another Simulation Add-In [33] - Free Monte Carlo Simulation Add-In for Excel created by Rutgers University
- MonteCarlito [34] - Open-source Monte Carlo Add-In for Excel (GPL-licensed)
- STMC [35] - Statistics and Monte Carlo (STMC) Fortran 77 package.
- Easy Monte Carlo Tool [36] - Simple free MS Excel tool created by UK Government for children's services commissioning
- Gambet [37] - Monte Carlo simulation of x-ray and electron transport in matter with 3D electric and magnetic fields by Field Precision LLC.

References

[1] Douglas Hubbard "How to Measure Anything: Finding the Value of Intangibles in Business" pg. 46, John Wiley & Sons, 2007
[2] Douglas Hubbard "The Failure of Risk Management: Why It's Broken and How to Fix It", John Wiley & Sons, 2009
[3] Nicholas Metropolis (1987). "The beginning of the Monte Carlo method" (http://library.lanl.gov/la-pubs/00326866.pdf). *Los Alamos Science* (1987 Special Issue dedicated to Stanislaw Ulam): 125–130. .
[4] http://people.cs.ubc.ca/~nando/papers/mlintro.pdf
[5] Charles Grinstead & J. Laurie Snell "Introduction to Probability" pp. 10-11, American Mathematical Society, 1997
[6] H.L. Anderson, "Metropolis, Monte Carlo and the MANIAC," (http://library.lanl.gov/cgi-bin/getfile?00326886.pdf) Los Alamos Science, no. 14, pp. 96-108, 1986.
[7] Stephan A. Baeurle (2009). "Multiscale modeling of polymer materials using field-theoretic methodologies: a survey about recent developments" (http://www.springerlink.com/content/xl057580272w8703/). *Journal of Mathematical Chemistry* **46** (2): 363–426. doi:10.1007/s10910-008-9467-3. .
[8] H. T. MacGillivray, R. J. Dodd, Monte-Carlo simulations of galaxy systems, *Astrophysics and Space Science*, Volume 86, Number 2 / September, 1982, Springer Netherlands (http://www.springerlink.com/content/rp3g1q05j176r108/fulltext.pdf)
[9] David Vose: "Risk Analysis, A Quantitative Guide," Second Edition, p. 13, John Wiley & Sons, 2000.
[10] Ibid, p. 16
[11] Ibid, p. 17, showing graph
[12] http://www.ipgp.jussieu.fr/~tarantola/Files/Professional/Papers_PDF/MonteCarlo_latex.pdf
[13] http://www.ipgp.jussieu.fr/~tarantola/Files/Professional/SIAM/index.html
[14] Davenport, J. H.. "Primality testing revisited" (http://doi.acm.org/10.1145/143242.143290). doi:. . Retrieved 2007-08-19.
[15] http://jstor.org/stable/2280232
[16] http://www.ipgp.jussieu.fr/~tarantola/Files/Professional/SIAM/index.html
[17] http://mathworld.wolfram.com/MonteCarloMethod.html
[18] http://www.phy.ornl.gov/csep/CSEP/MC/MC.html
[19] http://www.sitmo.com/eqcat/15
[20] http://www.chem.unl.edu/zeng/joy/mclab/mcintro.html
[21] http://office.microsoft.com/en-us/assistance/HA011118931033.aspx
[22] http://www.brighton-webs.co.uk/montecarlo/concept.asp
[23] http://www.cooper.edu/engineering/chemechem/monte.html
[24] http://www.princeton.edu/~achremos/Applet1-page.htm
[25] http://www.global-derivatives.com/maths/k-o.php
[26] http://twt.mpei.ac.ru/MAS/Worksheets/approxpi.mcd
[27] http://papers.ssrn.com/sol3/papers.cfm?abstract_id=265905
[28] http://en.wikiversity.org/wiki/Probabilistic_Assessment_of_Structures
[29] http://www.puc-rio.br/marco.ind/quasi_mc.html
[30] http://knol.google.com/k/giancarlo-vercellino/pricing-using-monte-carlo-simulation/11d5i2rgd9gn5/3#
[31] http://www.mrc-bsu.cam.ac.uk/bugs/
[32] http://www.statistics101.net
[33] http://yasai.rutgers.edu/

[34] http://www.montecarlito.com/
[35] http://www.worldscibooks.com/physics/5602.html
[36] http://www.dcsf.gov.uk/everychildmatters/resources-and-practice/IG00215/
[37] http://www.fieldp.com/gambet.html

Further Reading

NP-complete

In computational complexity theory, the complexity class **NP-complete** (abbreviated **NP-C** or **NPC**), is a class of problems having two properties:

- Any given solution to the problem can be *verified* quickly (in polynomial time); the set of problems with this property is called NP (nondeterministic polynomial time).
- If the problem can be *solved* quickly (in polynomial time), then so can every problem in NP.

Although any given solution to such a problem can be verified quickly, there is no known efficient way to locate a solution in the first place; indeed, the most notable characteristic of NP-complete problems is that no fast solution to them is known. That is, the time required to solve the problem using any currently known algorithm increases very quickly as the size of the problem grows. As a result, the time required to solve even moderately large versions of many of these problems easily reaches into the billions or trillions of years, using any amount of computing power available today. As a consequence, determining whether or not it is possible to solve these problems quickly is one of the principal unsolved problems in computer science today.

While a method for computing the solutions to NP-complete problems using a reasonable amount of time remains undiscovered, computer scientists and programmers still frequently encounter NP-complete problems. An expert programmer should be able to recognize an NP-complete problem so that he or she does not unknowingly spend time trying to solve a problem which so far has eluded generations of computer scientists. Instead, NP-complete problems are often addressed by using approximation algorithms.

Formal overview

NP-complete is a subset of NP, the set of all decision problems whose solutions can be verified in polynomial time; *NP* may be equivalently defined as the set of decision problems that can be solved in polynomial time on a nondeterministic Turing machine. A problem *p* in NP is also in NPC if and only if every other problem in NP can be transformed into *p* in polynomial time. NP-complete can also be used as an adjective: problems in the class NP-complete are known as NP-complete problems.

NP-complete problems are studied because the ability to quickly verify solutions to a problem (NP) seems to correlate with the ability to quickly solve that problem (P). It is not known whether every problem in NP can be quickly solved—this is called the P = NP problem. But if *any single problem* in NP-complete can be solved quickly, then *every problem in NP* can also be quickly solved, because the definition of an NP-complete problem states that every problem in NP must be quickly reducible to every problem in NP-complete (that is, it can be reduced in polynomial time). Because of this, it is often said that the NP-complete problems are *harder* or *more difficult* than NP problems in general.

Formal definition of NP-completeness

A decision problem *C* is NP-complete if:

1. *C* is in NP, and
2. Every problem in NP is reducible to *C* in polynomial time.

C can be shown to be in NP by demonstrating that a candidate solution to *C* can be verified in polynomial time.

A problem *K* is reducible to *C* if there is a polynomial-time many-one reduction, a deterministic algorithm which transforms any instance $k \in K$ into an instance $c \in C$, such that the answer to *c* is *yes* if and only if the answer to *k* is *yes*. To prove that an NP problem *C* is in fact an NP-complete problem it is sufficient to show that an already known NP-complete problem reduces to *C*.

Note that a problem satisfying condition 2 is said to be NP-hard, whether or not it satisfies condition 1.

A consequence of this definition is that if we had a polynomial time algorithm (on a UTM, or any other Turing-equivalent abstract machine) for *C*, we could solve all problems in NP in polynomial time.

Background

The concept of *NP-complete* was introduced in 1971 by Stephen Cook in a paper entitled *The complexity of theorem-proving procedures* on pages 151-158 of the *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*, though the term *NP-complete* did not appear anywhere in his paper. At that computer science conference, there was a fierce debate among the computer scientists about whether NP-complete problems could be solved in polynomial time on a deterministic Turing machine. John Hopcroft brought everyone at the conference to a consensus that the question of whether NP-complete problems are solvable in polynomial time should be put off to be solved at some later date, since nobody had any formal proofs for their claims one way or the other. This is known as the question of whether P=NP.

Nobody has yet been able to determine conclusively whether NP-complete problems are in fact solvable in polynomial time, making this one of the great unsolved problems of mathematics. The Clay Mathematics Institute is offering a US$1 million reward to anyone who has a formal proof that P=NP or that P≠NP.

In the celebrated Cook-Levin theorem (independently proved by Leonid Levin), Cook proved that the Boolean satisfiability problem is NP-complete (a simpler, but still highly technical proof of this is available). In 1972, Richard Karp proved that several other problems were also NP-complete (see Karp's 21 NP-complete problems); thus there is a class of NP-complete problems (besides the Boolean satisfiability problem). Since Cook's original results, thousands of other problems have been shown to be NP-complete by reductions from other problems previously shown to be NP-complete; many of these problems are collected in Garey and Johnson's 1979 book *Computers and Intractability: A Guide to NP-Completeness*.

NP-complete problems

An interesting example is the graph isomorphism problem, the graph theory problem of determining whether a graph isomorphism exists between two graphs. Two graphs are isomorphic if one can be transformed into the other simply by renaming vertices. Consider these two problems:

- Graph Isomorphism: Is graph G_1 isomorphic to graph G_2?
- Subgraph Isomorphism: Is graph G_1 isomorphic to a subgraph of graph G_2?

The Subgraph Isomorphism problem is NP-complete. The graph isomorphism problem is suspected to be neither in P nor NP-complete, though it is in NP. This is an example of a problem that is thought to be **hard**, but isn't thought to be NP-complete.

The easiest way to prove that some new problem is NP-complete is first to prove that it is in NP, and then to reduce some known NP-complete problem to it. Therefore, it is useful to know a variety of NP-complete problems. The list

below contains some well-known problems that are NP-complete when expressed as decision problems.

- Boolean satisfiability problem (Sat.)
- N-puzzle
- Knapsack problem
- Hamiltonian path problem
- Travelling salesman problem
- Subgraph isomorphism problem
- Subset sum problem
- Clique problem
- Vertex cover problem
- Independent set problem
- Dominating set problem
- Graph coloring problem

To the right is a diagram of some of the problems and the reductions typically used to prove their NP-completeness. In this diagram, an arrow from one problem to another indicates the direction of the reduction. Note that this diagram is misleading as a description of the mathematical relationship between these problems, as there exists a polynomial-time reduction between any two NP-complete problems; but it indicates where demonstrating this polynomial-time reduction has been easiest.

There is often only a small difference between a problem in P and an NP-complete problem. For example, the 3-satisfiability problem, a restriction of the boolean satisfiability problem, remains NP-complete, whereas the slightly more restricted 2-satisfiability problem is in P (specifically, NL-complete), and the slightly more general max. 2-sat. problem is again NP-complete. Determining whether a graph can be colored with 2 colors is in P, but with 3 colors is NP-complete, even when restricted to planar graphs. Determining if a graph is a cycle or is bipartite is very easy (in L), but finding a maximum bipartite or a maximum cycle subgraph is NP-complete. A solution of the knapsack problem within any fixed percentage of the optimal solution can be computed in polynomial time, but finding the optimal solution is NP-complete.

Solving NP-complete problems

At present, all known algorithms for NP-complete problems require time that is superpolynomial in the input size, and it is unknown whether there are any faster algorithms.

The following techniques can be applied to solve computational problems in general, and they often give rise to substantially faster algorithms:

- Approximation: Instead of searching for an optimal solution, search for an "almost" optimal one.
- Randomization: Use randomness to get a faster average running time, and allow the algorithm to fail with some small probability. See Monte Carlo method.
- Restriction: By restricting the structure of the input (e.g., to planar graphs), faster algorithms are usually possible.
- Parameterization: Often there are fast algorithms if certain parameters of the input are fixed.
- Heuristic: An algorithm that works "reasonably well" in many cases, but for which there is no proof that it is both always fast and always produces a good result. Metaheuristic approaches are often used.

One example of a heuristic algorithm is a suboptimal $O(n \log n)$ greedy coloring algorithm used for graph coloring during the register allocation phase of some compilers, a technique called graph-coloring global register allocation. Each vertex is a variable, edges are drawn between variables which are being used at the same time, and colors indicate the register assigned to each variable. Because most RISC machines have a fairly large number of general-purpose registers, even a heuristic approach is effective for this application.

Completeness under different types of reduction

In the definition of NP-complete given above, the term *reduction* was used in the technical meaning of a polynomial-time many-one reduction.

Another type of reduction is polynomial-time Turing reduction. A problem *X* is polynomial-time Turing-reducible to a problem *Y* if, given a subroutine that solves *Y* in polynomial time, one could write a program that calls this subroutine and solves *X* in polynomial time. This contrasts with many-one reducibility, which has the restriction that the program can only call the subroutine once, and the return value of the subroutine must be the return value of the program.

If one defines the analogue to NP-complete with Turing reductions instead of many-one reductions, the resulting set of problems won't be smaller than NP-complete; it is an open question whether it will be any larger. If the two concepts were the same, then it would follow that NP = co-NP. This holds because by their definition the classes of NP-complete and co-NP-complete problems under Turing reductions are the same and because these classes are both supersets of the same classes defined with many-one reductions. So if both definitions of NP-completeness are equal then there is a co-NP-complete problem (under both definitions) such as for example the complement of the boolean satisfiability problem that is also NP-complete (under both definitions). This implies that NP = co-NP as is shown in the proof in the co-NP article. Although whether NP = co-NP is an open question it is considered unlikely and therefore it is also unlikely that the two definitions of NP-completeness are equivalent.

Another type of reduction that is also often used to define NP-completeness is the logarithmic-space many-one reduction which is a many-one reduction that can be computed with only a logarithmic amount of space. Since every computation that can be done in logarithmic space can also be done in polynomial time it follows that if there is a logarithmic-space many-one reduction then there is also a polynomial-time many-one reduction. This type of reduction is more refined than the more usual polynomial-time many-one reductions and it allows us to distinguish more classes such as P-complete. Whether under these types of reductions the definition of NP-complete changes is still an open problem.

See also

- List of NP-complete problems
- Almost complete
- Ladner's theorem
- Strongly NP-complete
- P = NP problem
- NP-hard

References

- Garey, M.R.; Johnson, D.S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York: W.H. Freeman. ISBN 0-7167-1045-5. This book is a classic, developing the theory, then cataloguing *many* NP-Complete problems.
- Cook, S.A. (1971). "The complexity of theorem proving procedures". *Proceedings, Third Annual ACM Symposium on the Theory of Computing, ACM, New York*. pp. 151–158. doi:10.1145/800157.805047.
- Dunne, P.E. "An annotated list of selected NP-complete problems" [1]. COMP202, Dept. of Computer Science, University of Liverpool. Retrieved 2008-06-21.
- Crescenzi, P.; Kann, V.; Halldórsson, M.; Karpinski, M.; Woeginger, G. "A compendium of NP optimization problems" [2]. KTH NADA, Stockholm. Retrieved 2008-06-21.
- Dahlke, K. "NP-complete problems" [3]. *Math Reference Project*. Retrieved 2008-06-21.

- Karlsson, R. "Lecture 8: NP-complete problems" [4] (PDF). Dept. of Computer Science, Lund University, Sweden. Retrieved 2008-06-21.
- Sun, H.M. "The theory of NP-completeness" [5] (PPT). Information Security Laboratory, Dept. of Computer Science, National Tsing Hua University, Hsinchu City, Taiwan. Retrieved 2008-06-21.
- Jiang, J.R. "The theory of NP-completeness" [6] (PPT). Dept. of Computer Science and Information Engineering, National Central University, Jhongli City, Taiwan. Retrieved 2008-06-21.
- Cormen, T.H.; Leiserson, C.E., Rivest, R.L.; Stein, C. (2001). *Introduction to Algorithms* (2nd ed.). MIT Press and McGraw-Hill. Chapter 34: NP–Completeness, pp. 966–1021. ISBN 0-262-03293-7.
- Sipser, M. (1997). *Introduction to the Theory of Computation*. PWS Publishing. Sections 7.4–7.5 (NP-completeness, Additional NP-complete Problems), pp. 248–271.
- Papadimitriou, C. (1994). *Computational Complexity* (1st ed.). Addison Wesley. Chapter 9 (NP-complete problems), pp. 181–218. ISBN 0201530821.
- Computational Complexity of Games and Puzzles [7]
- Tetris is Hard, Even to Approximate [8]
- Minesweeper is NP-complete! [9]

Further reading

- Scott Aaronson, *NP-complete Problems and Physical Reality* [10], ACM SIGACT News, Vol. 36, No. 1. (March 2005), pp. 30-52.
- Lance Fortnow, *The status of the P versus NP problem* [11], Commun. ACM, Vol. 52, No. 9. (2009), pp. 78-86.

References

[1] http://www.csc.liv.ac.uk/~ped/teachadmin/COMP202/annotated_np.html
[2] http://www.nada.kth.se/~viggo/problemlist/compendium.html
[3] http://www.mathreference.com/lan-cx-np,intro.html
[4] http://www.cs.lth.se/home/Rolf_Karlsson/bk/lect8.pdf
[5] http://is.cs.nthu.edu.tw/course/2008Spring/cs431102/hmsunCh08.ppt
[6] http://www.csie.ncu.edu.tw/%7Ejrjiang/alg2006/NPC-3.ppt
[7] http://www.ics.uci.edu/~eppstein/cgt/hard.html
[8] http://arxiv.org/abs/cs.CC/0210020
[9] http://for.mat.bham.ac.uk/R.W.Kaye/minesw/ordmsw.htm
[10] http://arxiv.org/abs/quant-ph/0502072
[11] http://people.cs.uchicago.edu/~fortnow/papers/pnp-cacm.pdf

Dynamic programming

In mathematics and computer science, **dynamic programming** is a method of solving complex problems by breaking them down into simpler steps. It is applicable to problems that exhibit the properties of overlapping subproblems which are only slightly smaller[1] and optimal substructure (described below). When applicable, the method takes much less time than naive methods.

Top-down dynamic programming simply means storing the results of certain calculations, which are then re-used later because the same calculation is a sub-problem in a larger calculation. Bottom-up dynamic programming involves formulating a complex calculation as a recursive series of simpler calculations.

History

The term *dynamic programming* was originally used in the 1940s by Richard Bellman to describe the process of solving problems where one needs to find the best decisions one after another. By 1953, he had refined this to the modern meaning, which refers specifically to nesting smaller decision problems inside larger decisions,[2] and the field was thereafter recognized by the IEEE as a systems analysis and engineering topic. Bellman's contribution is remembered in the name of the Bellman equation, a central result of dynamic programming which restates an optimization problem in recursive form.

The word *dynamic* was chosen by Bellman because it sounded impressive, not because it described how the method worked.[3] The word *programming* referred to the use of the method to find an optimal *program*, in the sense of a military schedule for training or logistics. This usage is the same as that in the phrases *linear programming* and *mathematical programming*, a synonym for optimization.[4]

Overview

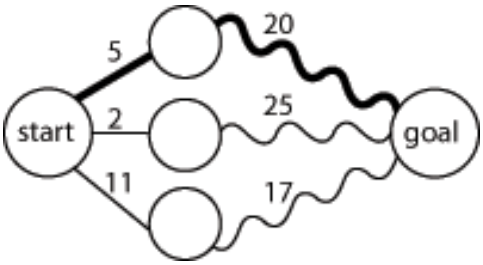


Figure 1. Finding the shortest path in a graph using optimal substructure; a straight line indicates a single edge; a wavy line indicates a shortest path between the two vertices it connects (other nodes on these paths are not shown); the bold line is the overall shortest path from start to goal.

Dynamic programming is both a mathematical optimization method, and a computer programming method. In both contexts, it refers to simplifying a complicated problem by breaking it down into simpler subproblems in a recursive manner. While some decision problems cannot be taken apart this way, decisions that span several points in time do often break apart recursively; Bellman called this the "Principle of Optimality". Likewise, in computer science, a problem which can be broken down recursively is said to have optimal substructure.

If subproblems can be nested recursively inside larger problems, so that dynamic programming methods are applicable, then there is a relation between the value of the larger problem and the values of the subproblems.[5] In the optimization literature this relationship is called the Bellman equation.

Dynamic programming in mathematical optimization

In terms of mathematical optimization, dynamic programming usually refers to a simplification of a decision by breaking it down into a sequence of decision steps over time. This is done by defining a sequence of **value functions** V_1 , V_2 , ... V_n , with an argument *y* representing the **state** of the system at times *i* from 1 to *n*. The definition of $V_n(y)$ is the value obtained in state *y* at the last time *n*. The values V_i at earlier times *i=n-1,n-2,...,2,1* can be found by working backwards, using a recursive relationship called the Bellman equation. For i=2,...n, V_{i-1} at any state y is

calculated from V_i by maximizing a simple function (usually the sum) of the gain from decision *i-1* and the function V_i at the new state of the system if this decision is made. Since V_i has already been calculated, for the needed states, the above operation yields V_{i-1} for all the needed states. Finally, V_1 at the initial state of the system is the value of the optimal solution. The optimal values of the decision variables can be recovered, one by one, by tracking back the calculations already performed.

Dynamic programming in computer programming

There are two key attributes that a problem must have in order for dynamic programming to be applicable: optimal substructure and overlapping subproblems which are only slightly smaller. When the overlapping problems are, say, half the size of the original problem the strategy is called "divide and conquer" rather than "dynamic programming". This is why merge sort, and quick sort, and finding all matches of a regular expression are not classified as dynamic programming problems.

Optimal substructure means that the solution to a given optimization problem can be obtained by the combination of optimal solutions to its subproblems. Consequently, the first step towards devising a dynamic programming solution is to check whether the problem exhibits such optimal substructure. Such optimal substructures are usually described by means of recursion. For example, given a graph *G=(V,E)*, the shortest path *p* from a vertex *u* to a vertex *v* exhibits optimal substructure: take any intermediate vertex *w* on this shortest path *p*. If *p* is truly the shortest path, then the path p_1 from *u* to *w* and p_2 from *w* to *v* are indeed the shortest paths between the corresponding vertices (by the simple cut-and-paste argument described in CLRS). Hence, one can easily formulate the solution for finding shortest paths in a recursive manner, which is what the Bellman-Ford algorithm does.

Overlapping subproblems means that the space of subproblems must be small, that is, any recursive algorithm solving the problem should solve the same subproblems over and over, rather than generating new subproblems. For example, consider the recursive formulation for generating the Fibonacci series: $F_i = F_{i-1} + F_{i-2}$, with base case $F_1=F_2=1$. Then $F_{43} = F_{42} + F_{41}$, and $F_{42} = F_{41} + F_{40}$. Now F_{41} is being solved in the recursive subtrees of both F_{43} as well as F_{42}. Even though the total number of subproblems is actually small (only 43 of them), we end up solving the same problems over and over if we adopt a naive recursive solution such as this. Dynamic programming takes account of this fact and solves each subproblem only once. Note that the subproblems must be only **'slightly**' smaller (typically taken to mean a constant additive factor) than the larger problem; when they are a multiplicative factor smaller the problem is no longer classified as dynamic programming (otherwise mergesort and quicksort would be dynamic programming problems).

This can be achieved in either of two ways:

- *Top-down approach*: This is the direct fall-out of the recursive formulation of any problem. If the solution to any problem can be formulated recursively using the solution to its subproblems, and if its subproblems are overlapping, then one can easily memoize or store the solutions to the subproblems in a table. Whenever we attempt to solve a new subproblem, we first check the table to see if it is already solved. If a solution has been recorded, we can use it directly, otherwise we solve the subproblem and add its solution to the table.
- *Bottom-up approach*: This is the more interesting case. Once we formulate the solution to a problem recursively as in terms of its subproblems, we can try reformulating the problem in a bottom-up fashion: try solving the subproblems first and use their solutions to build-on and arrive at solutions to bigger subproblems. This is also usually done in a tabular form by iteratively generating solutions to bigger and bigger subproblems by using the solutions to small subproblems. For example, if we already know the values of F_{41} and F_{40}, we can directly calculate the value of F_{42}.

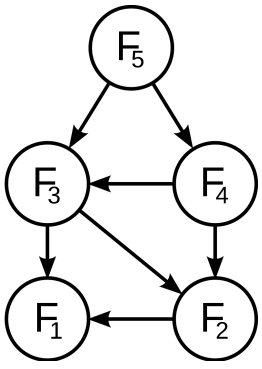


Figure 2. The subproblem graph for the Fibonacci sequence. The fact that it is not a tree indicates overlapping subproblems.

Some programming languages can automatically memoize the result of a function call with a particular set of arguments, in order to speed up call-by-name evaluation (this mechanism is referred to as *call-by-need*). Some languages make it possible portably (e.g. Scheme, Common Lisp or Perl), some need special extensions (e.g. C++, see [6]). Some languages have automatic memoization built in such as tabled Prolog. In any case, this is only possible for a referentially transparent function.

Example: mathematical optimization

Optimal consumption and saving

A mathematical optimization problem that is often used in teaching dynamic programming to economists (because it can be solved by hand[7]) concerns a consumer who lives over the periods $t = 0, 1, 2, ..., T$ and must decide how much to consume and how much to save in each period.

Let c_t be consumption in period t, and assume consumption yields utility $u(c_t) = \ln(c_t)$ as long as the consumer lives. Assume the consumer is impatient, so that he discounts future utility by a factor b each period, where $0 < b < 1$. Let k_t be capital in period t. Assume initial capital is a given amount $k_0 > 0$, and suppose that this period's capital and consumption determine next period's capital as $k_{t+1} = Ak_t^a - c_t$, where A is a positive constant and $0 < a < 1$. Assume capital cannot be negative. Then the consumer's decision problem can be written as follows:

$$\max \sum_{t=0}^{T} b^t \ln(c_t) \text{ subject to } k_{t+1} = Ak_t^a - c_t \geq 0 \text{ for all } t = 0, 1, 2, ..., T$$

Written this way, the problem looks complicated, because it involves solving for all the choice variables $c_0, c_1, c_2, ..., c_T$ and $k_1, k_2, k_3, ..., k_{T+1}$ simultaneously. (Note that k_0 is not a choice variable—the consumer's initial capital is taken as given.)

The dynamic programming approach to solving this problem involves breaking it apart into a sequence of smaller decisions. To do so, we define a sequence of *value functions* $V_t(k)$, for $t = 0, 1, 2, ..., T, T + 1$ which represent the value of having any amount of capital k at each time t. Note that $V_{T+1}(k) = 0$, that is, there is (by assumption) no utility from having capital after death.

The value of any quantity of capital at any previous time can be calculated by backward induction using the Bellman equation. In this problem, for each $t = 0, 1, 2, ..., T$, the Bellman equation is

$$V_t(k_t) = \max \left(\ln(c_t) + bV_{t+1}(k_{t+1})\right) \text{ subject to } k_{t+1} = Ak_t^a - c_t \geq 0$$

This problem is much simpler than the one we wrote down before, because it involves only two decision variables, c_t and k_{t+1}. Intuitively, instead of choosing his whole lifetime plan at birth, the consumer can take things one step at a time. At time t, his current capital k_t is given, and he only needs to choose current consumption c_t and saving k_{t+1}.

To actually solve this problem, we work backwards. For simplicity, the current level of capital is denoted as k. $V_{T+1}(k)$ is already known, so using the Bellman equation once we can calculate $V_T(k)$, and so on until we get to $V_0(k)$, which is the *value* of the initial decision problem for the whole lifetime. In other words, once we know $V_{T-j+1}(k)$, we can calculate $V_{T-j}(k)$, which is the maximum of $\ln(c_{T-j}) + bV_{T-j+1}(Ak^a - c_{T-j})$, where c_{T-j} is the variable and $Ak^a - c_{T-j} \geq 0$. It can be shown that the value function at time $t = T - j$ is

$$V_{T-j}(k) = a \sum_{i=0}^{j} a^i b^i \ln k + v_{T-j}$$

where each v_{T-j} is a constant, and the optimal amount to consume at time $t = T - j$ is

$$c_{T-j}(k) = \frac{1}{\sum_{i=0}^{j} a^i b^i} Ak^a$$

which can be simplified to

$$c_T(k) = Ak^a, \text{ and } c_{T-1}(k) = \frac{1}{1+ab}Ak^a, \text{ and } c_{T-2}(k) = \frac{1}{1+ab+a^2b^2}Ak^a, \text{ etcetera.}$$

We see that it is optimal to consume a larger fraction of current wealth as one gets older, finally consuming all current wealth in period T, the last period of life.

Examples: Computer algorithms

Fibonacci sequence

Here is a naive implementation of a function finding the *n*th member of the Fibonacci sequence, based directly on the mathematical definition:

```
function fib(n)
    if n = 0 return 0
    if n = 1 return 1
    return fib(n − 1) + fib(n − 2)
```

Notice that if we call, say, fib(5), we produce a call tree that calls the function on the same value many different times:

1. fib(5)
2. fib(4) + fib(3)
3. (fib(3) + fib(2)) + (fib(2) + fib(1))
4. ((fib(2) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))
5. (((fib(1) + fib(0)) + fib(1)) + (fib(1) + fib(0))) + ((fib(1) + fib(0)) + fib(1))

In particular, fib(2) was calculated three times from scratch. In larger examples, many more values of fib, or *subproblems*, are recalculated, leading to an exponential time algorithm.

Now, suppose we have a simple map object, *m*, which maps each value of fib that has already been calculated to its result, and we modify our function to use it and update it. The resulting function requires only O(*n*) time instead of exponential time:

```
var m := map(0 → 0, 1 → 1)
function fib(n)
    if map m does not contain key n
        m[n] := fib(n − 1) + fib(n − 2)
    return m[n]
```

This technique of saving values that have already been calculated is called *memoization*; this is the top-down approach, since we first break the problem into subproblems and then calculate and store values.

In the **bottom-up** approach we calculate the smaller values of fib first, then build larger values from them. This method also uses O(*n*) time since it contains a loop that repeats n − 1 times, however it only takes constant (O(1)) space, in contrast to the top-down approach which requires O(*n*) space to store the map.

```
function fib(n)
    var previousFib := 0, currentFib := 1
    if n = 0
        return 0
    else if n = 1
        return 1
```

```
        repeat n − 1 times
            var newFib := previousFib + currentFib
            previousFib := currentFib
            currentFib  := newFib
        return currentFib
```

In both these examples, we only calculate fib(2) one time, and then use it to calculate both fib(4) and fib(3), instead of computing it every time either of them is evaluated.

A type of balanced 0-1 matrix

Consider the problem of assigning values, either zero or one, to the positions of an *n x n* matrix, n even, so that each row and each column contains exactly $n/2$zeros and $n/2$ones. For example, when $n = 4$, three possible solutions are:

```
+ − − − − +              + − − − − +                  + − − − − +
| 0 1 0 1 |              | 0 0 1 1 |                  | 1 1 0 0 |
| 1 0 1 0 |      and     | 0 0 1 1 |        and       | 0 0 1 1 |
| 0 1 0 1 |              | 1 1 0 0 |                  | 1 1 0 0 |
| 1 0 1 0 |              | 1 1 0 0 |                  | 0 0 1 1 |
+ − − − − +              + − − − − +                  + − − − − +
```

We ask how many different assignments there are for a given n. There are at least three possible approaches: brute force, backtracking, and dynamic programming. Brute force consists of checking all assignments of zeros and ones and counting those that have balanced rows and columns ($n/2$zeros and $n/2$ones). As there are $\binom{n}{n/2}^n$ possible assignments, this strategy is not practical except maybe up to $n = 6$. Backtracking for this problem consists of choosing some order of the matrix elements and recursively placing ones or zeros, while checking that in every row and column the number of elements that have not been assigned plus the number of ones or zeros are both at least *n / 2*. While more sophisticated than brute force, this approach will visit every solution once, making it impractical for *n* larger than six, since the number of solutions is already 116963796250 for *n* = *8*, as we shall see. Dynamic programming makes it possible to count the number of solutions without visiting them all.

We consider $k \times n$ boards, where $1 \leq k \leq n$ whose k rows contain $n/2$zeros and $n/2$ones. The function *f* to which memoization is applied maps vectors of *n* pairs of integers to the number of admissible boards (solutions). There is one pair for each column and its two components indicate respectively the number of ones and zeros that have yet to be placed in that column. We seek the value of $f((n/2, n/2), (n/2, n/2), \ldots (n/2, n/2))$(n arguments or one vector of n elements). The process of subproblem creation involves iterating over every one of $\binom{n}{n/2}$ possible assignments for the top row of the board, and going through every column, subtracting one from the appropriate element of the pair for that column, depending on whether the assignment for the top row contained a zero or a one at that position. If any one of the results is negative, then the assignment is invalid and does not contribute to the set of solutions (recursion stops). Otherwise, we have an assignment for the top row of the $k \times n$ board and recursively compute the number of solutions to the remaining $(k - 1) \times n$ board, adding the numbers of solutions for every admissible assignment of the top row and returning the sum, which is being memoized. The base case is the trivial subproblem, which occurs for a $1 \times n$ board. The number of solutions for this board is either zero or one, depending on whether the vector is a permutation of $n/2$ $(0, 1)$and $n/2$ $(1, 0)$pairs or not.
For example, in the two boards shown above the sequences of vectors would be

```
((2, 2) (2, 2) (2, 2) (2, 2))        ((2, 2) (2, 2) (2, 2) (2, 2))      k
 = 4
```

```
  0      1      0      1              0      0      1      1

((1, 2) (2, 1) (1, 2) (2, 1))       ((1, 2) (1, 2) (2, 1) (2, 1))     k
 = 3
  1      0      1      0              0      0      1      1

((1, 1) (1, 1) (1, 1) (1, 1))       ((0, 2) (0, 2) (2, 0) (2, 0))     k
 = 2
  0      1      0      1              1      1      0      0

((0, 1) (1, 0) (0, 1) (1, 0))       ((0, 1) (0, 1) (1, 0) (1, 0))     k
 = 1
  1      0      1      0              1      1      0      0

((0, 0) (0, 0) (0, 0) (0, 0))       ((0, 0) (0, 0), (0, 0) (0, 0))
```

The number of solutions (sequence A058527 [8] in OEIS) is

$$1, 2, 90, 297200, 116963796250, 6736218287430460752, \ldots$$

Links to the Perl source of the backtracking approach, as well as a MAPLE and a C implementation of the dynamic programming approach may be found among the external links.

Checkerboard

Consider a checkerboard with $n \times n$ squares and a cost-function $c(i, j)$ which returns a cost associated with square i,j (i being the row, j being the column). For instance (on a 5 × 5 checkerboard),

5	6	7	4	7	8
4	7	6	1	1	4
3	3	5	7	8	2
2	-	6	7	0	-
1	-	-	5*	-	-
	1	**2**	**3**	**4**	**5**

Thus $c(1, 3) = 5$

Let us say you had a checker that could start at any square on the first rank (i.e., row) and you wanted to know the shortest path (sum of the costs of the visited squares are at a minimum) to get to the last rank, assuming the checker could move only diagonally left forward, diagonally right forward, or straight forward. That is, a checker on (1,3) can move to (2,2), (2,3) or (2,4).

5					
4					
3					
2		x	x	x	
1			o		
	1	**2**	**3**	**4**	**5**

This problem exhibits **optimal substructure**. That is, the solution to the entire problem relies on solutions to subproblems. Let us define a function $q(i, j)$ as

$q(i, j)$ = the minimum cost to reach square (i, j)

If we can find the values of this function for all the squares at rank n, we pick the minimum and follow that path backwards to get the shortest path.

Note that $q(i, j)$ is equal to the minimum cost to get to any of the three squares below it (since those are the only squares that can reach it) plus $c(i, j)$. For instance:

5					
4			A		
3		B	C	D	
2					
1					
	1	**2**	**3**	**4**	**5**

$$q(A) = min(q(B), q(C), q(D)) + c(A)$$

Now, let us define $q(i, j)$ in somewhat more general terms:

$$q(i,j) = \begin{cases} \infty & j < 1 \text{ or } j > n \\ c(i,j) & i = 1 \\ \min(q(i-1,j-1), q(i-1,j), q(i-1,j+1)) + c(i,j) & \text{otherwise.} \end{cases}$$

The first line of this equation is there to make the recursive property simpler (when dealing with the edges, so we need only one recursion). The second line says what happens in the last rank, to provide a base case. The third line, the recursion, is the important part. It is similar to the A,B,C,D example. From this definition we can make a straightforward recursive code for $q(i, j)$. In the following pseudocode, n is the size of the board, c(i, j) is the cost-function, and min() returns the minimum of a number of values:

```
function minCost(i, j)
    if j < 1 or j > n
        return infinity
    else if i = 5
        return c(i, j)
    else
        return min( minCost(i+1, j-1), minCost(i+1, j), minCost(i+1,
j+1) ) + c(i, j)
```

It should be noted that this function only computes the path-cost, not the actual path. We will get to the path soon. This, like the Fibonacci-numbers example, is horribly slow since it spends mountains of time recomputing the same shortest paths over and over. However, we can compute it much faster in a bottom-up fashion if we store path-costs

in a two-dimensional array q[i, j] rather than using a function. This avoids recomputation; before computing the cost of a path, we check the array q[i, j] to see if the path cost is already there.

We also need to know what the actual shortest path is. To do this, we use another array p[i, j], a *predecessor array*. This array implicitly stores the path to any square *s* by storing the previous node on the shortest path to *s*, i.e. the predecessor. To reconstruct the path, we lookup the predecessor of *s*, then the predecessor of that square, then the predecessor of that square, and so on, until we reach the starting square. Consider the following code:

```
function computeShortestPathArrays()
    for x from 1 to n
        q[1, x] := c(1, x)
    for y from 1 to n
        q[y, 0]     := infinity
        q[y, n + 1] := infinity
    for y from 2 to n
        for x from 1 to n
            m := min(q[y-1, x-1], q[y-1, x], q[y-1, x+1])
            q[y, x] := m + c(y, x)
            if m = q[y-1, x-1]
                p[y, x] := -1
            else if m = q[y-1, x]
                p[y, x] :=  0
            else
                p[y, x] :=  1
```

Now the rest is a simple matter of finding the minimum and printing it.

```
function computeShortestPath()
    computeShortestPathArrays()
    minIndex := 1
    min := q[n, 1]
    for i from 2 to n
        if q[n, i] < min
            minIndex := i
            min := q[n, i]
    printPath(n, minIndex)
```

```
function printPath(y, x)
    print(x)
    print("<-")
    if y = 2
        print(x + p[y, x])
    else
        printPath(y-1, x + p[y, x])
```

Sequence alignment

In genetics, sequence alignment is an important application where dynamic programming is essential.[3] Typically, the problem consists of transforming one sequence into another using edit operations that replace, insert, or remove an element. Each operation has an associated cost, and the goal is to find the sequence of edits with the lowest total cost.

The problem can be stated naturally as a recursion, a sequence A is optimally edited into a sequence B by either:

1. inserting the first character of B, and performing an optimal alignment of A and the tail of B
2. deleting the first character of A, and performing the optimal alignment of the tail of A and B
3. replacing the first character of A with the first character of B, and performing optimal alignments of the tails of A and B.

The partial alignments can be tabulated in a matrix, where cell (i,j) contains the cost of the optimal alignment of A[1..i] to B[1..j]. The cost in cell (i,j) can be calculated by adding the cost of the relevant operations to the cost of its neighboring cells, and selecting the optimum.

Different variants exist, see Smith-Waterman and Needleman-Wunsch.

Algorithms that use dynamic programming

- Backward induction as a solution method for finite-horizon discrete-time dynamic optimization problems
- Method of undetermined coefficients can be used to solve the Bellman equation in infinite-horizon, discrete-time, discounted, time-invariant dynamic optimization problems
- Many string algorithms including longest common subsequence, longest increasing subsequence, longest common substring, Levenshtein distance (edit distance).
- Many algorithmic problems on graphs can be solved efficiently for graphs of bounded treewidth or bounded clique-width by using dynamic programming on a tree decomposition of the graph.
- The Cocke-Younger-Kasami (CYK) algorithm which determines whether and how a given string can be generated by a given context-free grammar
- The use of transposition tables and refutation tables in computer chess
- The Viterbi algorithm (used for hidden Markov models)
- The Earley algorithm (a type of chart parser)
- The Needleman-Wunsch and other algorithms used in bioinformatics, including sequence alignment, structural alignment, RNA structure prediction.
- Floyd's All-Pairs shortest path algorithm
- Optimizing the order for chain matrix multiplication
- Pseudopolynomial time algorithms for the Subset Sum and Knapsack and Partition problem Problems
- The dynamic time warping algorithm for computing the global distance between two time series
- The Selinger (a.k.a. System R) algorithm for relational database query optimization
- De Boor algorithm for evaluating B-spline curves
- Duckworth-Lewis method for resolving the problem when games of cricket are interrupted
- The Value Iteration method for solving Markov decision processes
- Some graphic image edge following selection methods such as the "magnet" selection tool in Photoshop
- Some methods for solving interval scheduling problems
- Some methods for solving word wrap problems
- Some methods for solving the travelling salesman problem, either exactly (in exponential time) or approximately (e.g. via the bitonic tour)
- Recursive least squares method
- Beat tracking in Music Information Retrieval.
- Adaptive Critic training strategy for artificial neural networks

- Stereo algorithms for solving the Correspondence problem used in stereo vision.
- Seam carving (content aware image resizing)
- The Bellman-Ford algorithm for finding the shortest distance in a graph.
- Some approximate solution methods for the linear search problem.

See also

- Bellman equation
- Divide and conquer algorithm
- Greedy algorithm
- Markov Decision Process
- Stochastic programming

Further reading

- Adda, Jerome; Cooper, Russell (2003), *Dynamic Economics* [9], MIT Press. An accessible introduction to dynamic programming in economics. The link contains sample programs.
- Bellman, Richard (1957), *Dynamic Programming*, Princeton University Press. Dover paperback edition (2003), ISBN 0486428095.
- Bertsekas, D. P. (2000), *Dynamic Programming and Optimal Control* (2nd ed.), Athena Scientific, ISBN 1-886529-09-4. In two volumes.
- Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2001), *Introduction to Algorithms* (2nd ed.), MIT Press & McGraw-Hill, ISBN 0-262-03293-7. Especially pp. 323–69.
- Dreyfus, Stuart E.; Law, Averill M. (1977), *The art and theory of dynamic programming*, Academic Press, ISBN 978-0122218606.
- Giegerich, R.; Meyer, C.; Steffen, P. (2004), "A Discipline of Dynamic Programming over Sequence Data" [10], *Science of Computer Programming* **51** (3): 215–263, doi:10.1016/j.scico.2003.12.005.
- Meyn, Sean (2007), *Control Techniques for Complex Networks* [11], Cambridge University Press, ISBN 9780521884419.
- S. S. Sritharan, "Dynamic Programming of the Navier-Stokes Equations," in Systems and Control Letters, Vol. 16, No. 4, 1991, pp. 299-307.
- Stokey, Nancy; Lucas, Robert E.; Prescott, Edward (1989), *Recursive Methods in Economic Dynamics*, Harvard Univ. Press, ISBN 9780674750968.

External links

- An Introduction to Dynamic Programming [12]
- Dyna [13], a declarative programming language for dynamic programming algorithms
- Wagner, David B., 1995, "Dynamic Programming. [14]" An introductory article on dynamic programming in Mathematica.
- Ohio State University: CIS 680: class notes on dynamic programming [15], by Eitan M. Gurari
- A Tutorial on Dynamic programming [16]
- MIT course on algorithms [17] - Includes a video lecture on DP along with lecture notes
- More DP Notes [18]
- King, Ian, 2002 (1987), "A Simple Introduction to Dynamic Programming in Macroeconomic Models. [19]" An introduction to dynamic programming as an important tool in economic theory.
- Dynamic Programming: from novice to advanced [20] A TopCoder.com article by Dumitru on Dynamic Programming

- Algebraic Dynamic Programming [21] - a formalized framework for dynamic programming, including an entry-level course [22] to DP, University of Bielefeld
- Dreyfus, Stuart, "Richard Bellman on the birth of Dynamic Programming. [23]„
- Dynamic programming tutorial [24]
- A Gentle Introduction to Dynamic Programming and the Viterbi Algorithm [25]
- Tabled Prolog BProlog [26] and XSB [27]

References

[1] S. Dasgupta, C.H. Papadimitriou, and U.V. Vazirani, '**Algorithms**', p173, available at http://www.cs.berkeley.edu/~vazirani/algorithms.html

[2] http://www.wu-wien.ac.at/usr/h99c/h9951826/bellman_dynprog.pdf

[3] Eddy, S. R., What is dynamic programming?, Nature Biotechnology, 22, 909-910 (2004).

[4] Nocedal, J.; Wright, S. J.: Numerical Optimization, page 9, Springer, 2006..

[5] Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; Stein, C. (2001), Introduction to Algorithms (2nd ed.), MIT Press & McGraw-Hill, ISBN 0-262-03293-7 . pp. 327–8.

[6] http://www.apl.jhu.edu/~paulmac/c++-memoization.html

[7] Stokey et al., 1989, Chap. 1

[8] http://en.wikipedia.org/wiki/Oeis%3Aa058527

[9] http://www.eco.utexas.edu/~cooper/dynprog/dynprog1.html

[10] http://bibiserv.techfak.uni-bielefeld.de/adp/ps/GIE-MEY-STE-2004.pdf

[11] https://netfiles.uiuc.edu/meyn/www/spm_files/CTCN/CTCN.html

[12] http://20bits.com/articles/introduction-to-dynamic-programming/

[13] http://www.dyna.org

[14] http://citeseer.ist.psu.edu/268391.html

[15] http://www.cse.ohio-state.edu/~gurari/course/cis680/cis680Ch21.html

[16] http://mat.gsia.cmu.edu/classes/dynamic/dynamic.html

[17] http://ocw.mit.edu/OcwWeb/Electrical-Engineering-and-Computer-Science/6-046JFall-2005/VideoLectures/detail/embed15.htm

[18] http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Dynamic

[19] http://researchspace.auckland.ac.nz/bitstream/handle/2292/190/230.pdf

[20] http://www.topcoder.com/tc?module=Static&d1=tutorials&d2=dynProg

[21] http://bibiserv.techfak.uni-bielefeld.de/adp/

[22] http://bibiserv.techfak.uni-bielefeld.de/dpcourse

[23] http://www.eng.tau.ac.il/~ami/cd/or50/1526-5463-2002-50-01-0048.pdf

[24] http://www.avatar.se/lectures/molbioinfo2001/dynprog/dynamic.html

[25] http://www.cambridge.org/resources/0521882672/7934_kaeslin_dynpro_new.pdf

[26] http://www.probp.com

[27] http://xsb.sourceforge.net/

Linear programming

Linear programming (LP) is a mathematical method for determining a way to achieve the best outcome (such as maximum profit or lowest cost) in a given mathematical model for some list of requirements represented as linear equations.

More formally, linear programming is a technique for the optimization of a linear objective function, subject to linear equality and linear inequality constraints. Given a polytope and a real-valued affine function defined on this polytope, a linear programming method will find a point on the polytope where this function has the smallest (or largest) value if such point exists, by searching through the polytope vertices.

Linear programs are problems that can be expressed in canonical form:

Maximize: $\mathbf{c}^{\mathrm{T}}\mathbf{x}$

Subject to: $A\mathbf{x} \leq \mathbf{b}$.

where $\mathbf{x}$ represents the vector of variables (to be determined), $\mathbf{c}$ and $\mathbf{b}$ are vectors of (known) coefficients and A is a (known) matrix of coefficients. The expression to be maximized or minimized is called the objective function ($\mathbf{c}^{\mathrm{T}}\mathbf{x}$ in this case). The equations $A\mathbf{x} \leq \mathbf{b}$ are the constraints which specify a convex polytope over which the objective function is to be optimized. (In this context, two vectors are comparable when every entry in one is less-than or equal-to the corresponding entry in the other. Otherwise, they are incomparable.)

Linear programming can be applied to various fields of study. It is used most extensively in business and economics, but can also be utilized for some engineering problems. Industries that use linear programming models include transportation, energy, telecommunications, and manufacturing. It has proved useful in modeling diverse types of problems in planning, routing, scheduling, assignment, and design.

History of linear programming

The problem of solving a system of linear inequalities dates back at least as far as Fourier, after whom the method of Fourier-Motzkin elimination is named. Linear programming arose as a mathematical model developed during the second world war to plan expenditures and returns in order to reduce costs to the army and increase losses to the enemy. It was kept secret until 1947. Postwar, many industries found its use in their daily planning.

The founders of the subject are Leonid Kantorovich, a Russian mathematician who developed linear programming problems in 1939, George B. Dantzig, who published the simplex method in 1947, and John von Neumann, who developed the theory of the duality in the same year. The linear programming problem was first shown to be solvable in polynomial time by Leonid Khachiyan in 1979, but a larger theoretical and practical breakthrough in the field came in 1984 when Narendra Karmarkar introduced a new interior point method for solving linear programming problems.

Dantzig's original example of finding the best assignment of 70 people to 70 jobs exemplifies the usefulness of linear programming. The computing power required to test all the permutations to select the best assignment is vast; the number of possible configurations exceeds the number of particles in the universe. However, it takes only a moment to find the optimum solution by posing the problem as a linear program and applying the Simplex algorithm. The theory behind linear programming drastically reduces the number of possible optimal solutions that must be checked.

Uses

Linear programming is a considerable field of optimization for several reasons. Many practical problems in operations research can be expressed as linear programming problems. Certain special cases of linear programming, such as *network flow* problems and *multicommodity flow* problems are considered important enough to have generated much research on specialized algorithms for their solution. A number of algorithms for other types of optimization problems work by solving LP problems as sub-problems. Historically, ideas from linear programming have inspired many of the central concepts of optimization theory, such as *duality, decomposition,* and the importance of *convexity* and its generalizations. Likewise, linear programming is heavily used in microeconomics and company management, such as planning, production, transportation, technology and other issues. Although the modern management issues are ever-changing, most companies would like to maximize profits or minimize costs with limited resources. Therefore, many issues can boil down to linear programming problems.

Standard form

Standard form is the usual and most intuitive form of describing a linear programming problem. It consists of the following three parts:

- A **linear function to be maximized**

  e.g., Maximize: $\mathbf{c}_1\mathbf{x}_1 + \mathbf{c}_2\mathbf{x}_2$
- **Problem constraints** of the following form

  e.g.,

  $$a_{1,1}\mathbf{x}_1 + a_{1,2}\mathbf{x}_2 \leq \mathbf{b}_1$$
  $$a_{2,1}\mathbf{x}_1 + a_{2,2}\mathbf{x}_2 \leq \mathbf{b}_2$$
  $$a_{3,1}\mathbf{x}_1 + a_{3,2}\mathbf{x}_2 \leq \mathbf{b}_3$$
- **Non-negative variables**

  e.g.,

  $$\mathbf{x}_1 \geq 0$$
  $$\mathbf{x}_2 \geq 0.$$

The problem is usually expressed in *matrix form*, and then becomes:

Maximize: $\mathbf{c}^{\mathrm{T}}\mathbf{x}$

Subject to: $A\mathbf{x} \leq \mathbf{b}, \mathbf{x} \geq 0.$

Other forms, such as minimization problems, problems with constraints on alternative forms, as well as problems involving negative variables can always be rewritten into an equivalent problem in standard form.

Example

Suppose that a farmer has a piece of farm land, say A square kilometres large, to be planted with either wheat or barley or some combination of the two. The farmer has a limited permissible amount F of fertilizer and P of insecticide which can be used, each of which is required in different amounts per unit area for wheat (F_1, P_1) and barley (F_2, P_2). Let S_1 be the selling price of wheat, and S_2 the price of barley. If we denote the area planted with wheat and barley by x_1 and x_2 respectively, then the optimal number of square kilometres to plant with wheat vs barley can be expressed as a linear programming problem:

$$\begin{aligned}
&\text{Maximize: } && S_1\mathbf{x}_1 + S_2\mathbf{x}_2 && \text{(maximize the revenue — revenue is the "objective function")}\\
&\text{Subject to:} && \mathbf{x}_1 + \mathbf{x}_2 \le A && \text{(limit on total area)}\\
& && F_1\mathbf{x}_1 + F_2\mathbf{x}_2 \le F && \text{(limit on fertilizer)}\\
& && P_1\mathbf{x}_1 + P_2\mathbf{x}_2 \le P && \text{(limit on insecticide)}\\
& && \mathbf{x}_1 \ge 0, \mathbf{x}_2 \ge 0 && \text{(cannot plant a negative area).}
\end{aligned}$$

Which in matrix form becomes:

$$\begin{aligned}
&\text{maximize } \begin{bmatrix} S_1 & S_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}\\
&\text{subject to } \begin{bmatrix} 1 & 1 \\ F_1 & F_2 \\ P_1 & P_2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \le \begin{bmatrix} A \\ F \\ P \end{bmatrix}, \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \ge 0.
\end{aligned}$$

Augmented form (slack form)

Linear programming problems must be converted into *augmented form* before being solved by the simplex algorithm. This form introduces non-negative *slack variables* to replace inequalities with equalities in the constraints. The problem can then be written in the following block matrix form:

Maximize *Z*:

$$\begin{bmatrix} 1 & -\mathbf{c}^T & 0 \\ 0 & \mathbf{A} & \mathbf{I} \end{bmatrix} \begin{bmatrix} Z \\ \mathbf{x} \\ \mathbf{x}_s \end{bmatrix} = \begin{bmatrix} 0 \\ \mathbf{b} \end{bmatrix}$$

$$\mathbf{x}, \mathbf{x}_s \ge 0$$

where $\mathbf{x}_s$ are the newly introduced slack variables, and *Z* is the variable to be maximized.

Example

The example above is converted into the following augmented form:

$$\begin{aligned}
&\text{Maximize: } && S_1\mathbf{x}_1 + S_2\mathbf{x}_2 && \text{(objective function)}\\
&\text{Subject to:} && \mathbf{x}_1 + \mathbf{x}_2 + \mathbf{x}_3 = A && \text{(augmented constraint)}\\
& && F_1\mathbf{x}_1 + F_2\mathbf{x}_2 + \mathbf{x}_4 = F && \text{(augmented constraint)}\\
& && P_1\mathbf{x}_1 + P_2\mathbf{x}_2 + \mathbf{x}_5 = P && \text{(augmented constraint)}\\
& && \mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4, \mathbf{x}_5 \ge 0. &&
\end{aligned}$$

where $\mathbf{x}_3$, $\mathbf{x}_4$, $\mathbf{x}_5$ are (non-negative) slack variables, representing in this example the unused area, the amount of unused fertilizer, and the amount of unused insecticide.

In matrix form this becomes:

Maximize *Z*:

$$\begin{bmatrix} 1 & -S_1 & -S_2 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & F_1 & F_2 & 0 & 1 & 0 \\ 0 & P_1 & P_2 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} Z \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} 0 \\ A \\ F \\ P \end{bmatrix}, \quad \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} \geq 0.$$

Duality

Every linear programming problem, referred to as a *primal* problem, can be converted into a dual problem, which provides an upper bound to the optimal value of the primal problem. In matrix form, we can express the *primal* problem as:

Maximize $\mathbf{c}^T\mathbf{x}$ subject to $A\mathbf{x} \leq \mathbf{b}$, $\mathbf{x} \geq 0$;

with the corresponding **symmetric** dual problem,

Minimize $\mathbf{b}^T\mathbf{y}$ subject to $A^T\mathbf{y} \geq \mathbf{c}$, $\mathbf{y} \geq 0$.

An alternative primal formulation is:

Maximize $\mathbf{c}^T\mathbf{x}$ subject to $A\mathbf{x} \leq \mathbf{b}$;

with the corresponding **asymmetric** dual problem,

Minimize $\mathbf{b}^T\mathbf{y}$ subject to $A^T\mathbf{y} = \mathbf{c}$, $\mathbf{y} \geq 0$.

There are two ideas fundamental to duality theory. One is the fact that (for the symmetric dual) the dual of a dual linear program is the original primal linear program. Additionally, every feasible solution for a linear program gives a bound on the optimal value of the objective function of its dual. The weak duality theorem states that the objective function value of the dual at any feasible solution is always greater than or equal to the objective function value of the primal at any feasible solution. The strong duality theorem states that if the primal has an optimal solution, $\mathbf{x}^*$, then the dual also has an optimal solution, $\mathbf{y}^*$, such that $\mathbf{c}^T\mathbf{x}^* = \mathbf{b}^T\mathbf{y}^*$.

A linear program can also be unbounded or infeasible. Duality theory tells us that if the primal is unbounded then the dual is infeasible by the weak duality theorem. Likewise, if the dual is unbounded, then the primal must be infeasible. However, it is possible for both the dual and the primal to be infeasible (See also Farkas' lemma).

Example

Revisit the above example of the farmer who may grow wheat and barley with the set provision of some A land, F fertilizer and P insecticide. Assume now that unit prices for each of these means of production (inputs) are set by a planning board. The planning board's job is to minimize the total cost of procuring the set amounts of inputs while providing the farmer with a floor on the unit price of each of his crops (outputs), S_1 for wheat and S_2 for barley. This corresponds to the following linear programming problem:

Minimize: $A\mathbf{y}_A + F\mathbf{y}_F + P\mathbf{y}_\mathrm{P}$ (minimize the total cost of the means of production as the "objective function")

Subject to: $\mathbf{y}_A + F_1\mathbf{y}_F + P_1\mathbf{y}_P \geq S_1$ (the farmer must receive no less than S_1 for his wheat)

$\mathbf{y}_A + F_2\mathbf{y}_F + P_2\mathbf{y}_P \geq S_2$ (the farmer must receive no less than S_2 for his barley)

$\mathbf{y}_A \geq 0, \mathbf{y}_F \geq 0, \mathbf{y}_P \geq 0$ (prices cannot be negative).

Which in matrix form becomes:

Minimize: $\begin{bmatrix} A & F & P \end{bmatrix} \begin{bmatrix} y_A \\ y_F \\ y_P \end{bmatrix}$

Subject to: $\begin{bmatrix} 1 & F_1 & P_1 \\ 1 & F_2 & P_2 \end{bmatrix} \begin{bmatrix} y_A \\ y_F \\ y_P \end{bmatrix} \geq \begin{bmatrix} S_1 \\ S_2 \end{bmatrix}, \begin{bmatrix} y_A \\ y_F \\ y_P \end{bmatrix} \geq 0.$

The primal problem deals with physical quantities. With all inputs available in limited quantities, and assuming the unit prices of all outputs is known, what quantities of outputs to produce so as to maximize total revenue? The dual problem deals with economic values. With floor guarantees on all output unit prices, and assuming the available quantity of all inputs is known, what input unit pricing scheme to set so as to minimize total expenditure?

To each variable in the primal space corresponds an inequality to satisfy in the dual space, both indexed by output type. To each inequality to satisfy in the primal space corresponds a variable in the dual space, both indexed by input type.

The coefficients that bound the inequalities in the primal space are used to compute the objective in the dual space, input quantities in this example. The coefficients used to compute the objective in the primal space bound the inequalities in the dual space, output unit prices in this example.

Both the primal and the dual problems make use of the same matrix. In the primal space, this matrix expresses the consumption of physical quantities of inputs necessary to produce set quantities of outputs. In the dual space, it expresses the creation of the economic values associated with the outputs from set input unit prices.

Since each inequality can be replaced by an equality and a slack variable, this means each primal variable corresponds to a dual slack variable, and each dual variable corresponds to a primal slack variable. This relation allows us to complementary slackness.

Another example

Sometimes, one may find it more intuitive to obtain the dual program without looking at program matrix. Consider the following linear program:

$$
\begin{aligned}
\text{minimize} \quad & \sum_{i=1}^{m} c_i x_i + \sum_{j=1}^{n} d_j t_j \\
\text{subject to} \quad & \sum_{i=1}^{m} a_{ij} x_i + e_j t_j \geq g_j \quad , \ 1 \leq j \leq n \\
& f_i x_i + \sum_{j=1}^{n} b_{ij} t_j \geq h_i \quad , \ 1 \leq i \leq m \\
& x_i \geq 0,\, t_j \geq 0 \quad , \ 1 \leq i \leq m, 1 \leq j \leq n
\end{aligned}
$$

We have $m + n$ conditions and all variables are non-negative. We shall define $m + n$ dual variables: $\mathbf{y}_j$ and $\mathbf{s}_i$. We get:

$$
\begin{aligned}
\text{minimize} \quad & \sum_{i=1}^{m} c_i x_i + \sum_{j=1}^{n} d_j t_j \\
\text{subject to} \quad & \sum_{i=1}^{m} a_{ij} x_i \cdot y_j + e_j t_j \cdot y_j \geq g_j \cdot y_j \quad , \ 1 \leq j \leq n \\
& f_i x_i \cdot s_i + \sum_{j=1}^{n} b_{ij} t_j \cdot s_i \geq h_i \cdot s_i \quad , \ 1 \leq i \leq m \\
& x_i \geq 0,\, t_j \geq 0 \quad , \ 1 \leq i \leq m, 1 \leq j \leq n \\
& y_j \geq 0,\, s_i \geq 0 \quad , \ 1 \leq j \leq n, 1 \leq i \leq m
\end{aligned}
$$

Since this is a minimization problem, we would like to obtain a dual program that is a lower bound of the primal. In other words, we would like the sum of all right hand side of the constraints to be the maximal under the condition that for each primal variable the sum of its coefficients do not exceed its coefficient in the linear function. For example, $\mathbf{x}_1$ appears in $n + 1$ constraints. If we sum its constraints' coefficients we get $a_{1,1}\mathbf{y}_1 + a_{1,2}\mathbf{y}_2 + ... + a_{1,n}\mathbf{y}_n + f_1\mathbf{s}_1$. This sum must be at most $\mathbf{c}_1$. As a result we get:

$$
\begin{aligned}
\text{maximize} \quad & \sum_{j=1}^{n} g_j y_j + \sum_{i=1}^{m} h_i s_i \\
\text{subject to} \quad & \sum_{j=1}^{n} a_{ij} y_j + f_i s_i \leq c_i \quad , \ 1 \leq i \leq m \\
& e_j y_j + \sum_{i=1}^{m} b_{ij} s_i \leq d_j \quad , \ 1 \leq j \leq n \\
& y_j \geq 0,\, s_i \geq 0 \quad , \ 1 \leq j \leq n, 1 \leq i \leq m
\end{aligned}
$$

Note that we assume in our calculations steps that the program is in standard form. However, any linear program may be transformed to standard form and it is therefore not a limiting factor.

Covering-packing dualities

Covering-packing dualities	
Covering problems	**Packing problems**
Minimum set cover	Maximum set packing
Minimum vertex cover	Maximum matching
Minimum edge cover	Maximum independent set

A covering LP is a linear program of the form:

Minimize: $\mathbf{b}^{\mathrm{T}}\mathbf{y}$,

Subject to: $A^{\mathrm{T}}\mathbf{y} \geq \mathbf{c}, \mathbf{y} \geq 0$,

such that the matrix A and the vectors $\mathbf{b}$ and $\mathbf{c}$ are non-negative.

The dual of a covering LP is a packing LP, a linear program of the form:

Maximize: $\mathbf{c}^{\mathrm{T}}\mathbf{x}$,

Subject to: $A\mathbf{x} \leq \mathbf{b}, \mathbf{x} \geq 0$,

such that the matrix A and the vectors $\mathbf{b}$ and $\mathbf{c}$ are non-negative.

Examples

Covering and packing LPs commonly arise as a linear programming relaxation of a combinatorial problem and are important in the study of approximation algorithms.[1] For example, the LP relaxations of the set packing problem, the independent set problem, and the matching problem are packing LPs. The LP relaxations of the set cover problem, the vertex cover problem, and the dominating set problem are covering LPs.

Finding a fractional coloring of a graph is another example of a covering LP. In this case, there is one constraint for each vertex of the graph and one variable for each independent set of the graph.

Complementary slackness

It is possible to obtain an optimal solution to the dual when only an optimal solution to the primal is known using the complementary slackness theorem. The theorem states:

Suppose that $\mathbf{x} = (\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n)$ is primal feasible and that $\mathbf{y} = (\mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_m)$ is dual feasible. Let $(\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_m)$ denote the corresponding primal slack variables, and let $(\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_n)$ denote the corresponding dual slack variables. Then $\mathbf{x}$ and $\mathbf{y}$ are optimal for their respective problems if and only if

- $\mathbf{x}_j\mathbf{z}_j = 0$, for $j = 1, 2, \ldots, n$, and
- $\mathbf{w}_i\mathbf{y}_i = 0$, for $i = 1, 2, \ldots, m$.

So if the i-th slack variable of the primal is not zero, then the i-th variable of the dual is equal zero. Likewise, if the j-th slack variable of the dual is not zero, then the j-th variable of the primal is equal to zero.

This necessary condition for optimality conveys a fairly simple economic principle. In standard form (when maximizing), if there is slack in a constrained primal resource (i.e., there are "leftovers"), then additional quantities of that resource must have no value. Likewise, if there is slack in the dual (shadow) price non-negativity constraint requirement , i.e., the price is not zero, then there must scarce supplies (no "leftovers").

Theory

Existence of optimal solutions

Geometrically, the linear constraints define a convex polytope, which is called the feasible region. A linear function is a convex function, which implies that every local minimum is a global minimum; similarly, a linear function is a concave function, which implies that every local maximum is a global maximum.

Optimal solution need not exist, for two reasons. First, if two constraints are inconsistent, then no feasible solution exists: For instance, the constraints $\mathbf{x} \geq 2$ and $\mathbf{x} \leq 1$) cannot be satisfied jointly; in this case, we say that the LP is *infeasible*. Second, when the polytope is unbounded in the direction of the gradient of the objective function (where the gradient of the objective function is the vector of the coeffients of the objective function), then no optimal value is attained.

Optimal vertices (and rays) of polyhedra

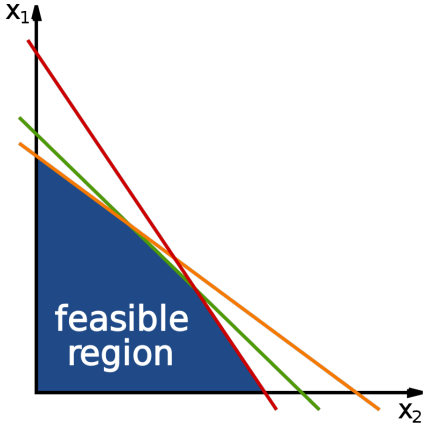
Otherwise, if a feasible solution exists and if the (linear) objective function is bounded, then the optimum value is always attained on the boundary of optimal level-set, by the *maximum principle* for *convex functions* (alternatively, by the *minimum* principle for *concave functions*): Recall that linear functions are both convex and concave. However, some problems have distinct optimal solutions: For example, the problem of finding a feasible solution to a system of linear inequalities is a linear programming problem in which the objective function is the zero function (that is, the the constant function taking the value zero everywhere): For this feasibility problem with the zero-function for its objective-function, if there are two distinct solutions, then every convex combination of the solutions is a solution.

The vertices of the polytope are also called *basic feasible solutions*. The reason for this choice of name is as follows. Let d denote the number of variables. Then the fundamental theorem of linear inequalities implies (for feasible problems) that for every vertex $\mathbf{x}^*$ of the LP feasible region, there exists a set of d (or fewer) inequality constraints from the LP such that, when we treat those d constraints as equalities, the unique solution is $\mathbf{x}^*$. Thereby we can study these vertices by means of looking at certain subsets of the set of all constraints (a discrete set), rather than the continuum of LP solutions. This principle underlies the simplex algorithm for solving linear programs.

Algorithms

The simplex algorithm of Dantzig

The simplex algorithm, developed by George Dantzig, solves LP problems by constructing a feasible solution at a vertex of the polytope and then walking along a path on the edges of the polytope to vertices with non-decreasing values of the objective function until an optimum is reached. In many practical problems, "stalling" occurs: Many pivots are made with no increase in the objective function.[2] In rare practical problems, the usual versions of the simplex algorithm may actually "cycle".[3] To avoid cycles, researchers developed new pivoting rules [4]



A series of linear constraints on two variables produces a region of possible values for those variables. Solvable problems will have a feasible region in the shape of a simple polygon.

In practice, the simplex algorithm is quite efficient and can be guaranteed to find the global optimum if certain precautions against *cycling* are taken. The simplex algorithm has been proved to solve

"random" problems efficiently, i.e. in a cubic number of steps (Borgwadt, Todd), which is similar to its behavior on practical problems[5]

However, the simplex algorithm has poor worst-case behavior: Klee and Minty constructed a family of linear programming problems for which the simplex method takes a number of steps exponential in the problem size.[6] In fact, for some time it was not known whether the linear programming problem was solvable in polynomial time (complexity class P).

The ellipsoid algorithm, following Khachiyan

This long standing issue was resolved by Leonid Khachiyan in 1979 with the introduction of the ellipsoid method, the first worst-case polynomial-time algorithm for linear programming. To solve a problem which has *n* variables and can be encoded in *L* input bits, this algorithm uses $O(n^4L)$ pseudo-arithmetic operations on numbers with $O(L)$ digits. Khahiyan's algorithm and his convergence analysis have (real-number) predecessors, notably the iterative methods developed by Naum Z. Shor and the approximation algorithms by Arkadi Nemirovski and D. Yudin.

Interior point methods, following Karmarkar

Khachiyan's algorithm was of landmark importance for establishing the polynomial-time solvability of linear programs. The algorithm had little practical impact, as the simplex method is more efficient for all but specially constructed families of linear programs. However, it inspired new lines of research in linear programming with the development of interior point methods, which can be implemented as a practical tool. In contrast to the simplex algorithm, which finds the optimal solution by progressing along points on the boundary of a polytopal set, interior point methods move through the interior of the feasible region.

In 1984, N. Karmarkar proposed a new interior point projective method for linear programming. Karmarkar's algorithm not only improved on Khachiyan's theoretical worst-case polynomial bound (giving $O(n^{3.5}L)$). Karmarkar also claimed that his algorithm exhibited dramatic practical performance improvements over the simplex method, which created great interest in interior-point methods. Since then, many interior point methods have been proposed and analyzed. Early successful implementations were based on *affine scaling* variants of the method. For both theoretical and practical properties, barrier function or path-following methods are the most common recently.

Comparison of interior-point methods versus simplex algorithms

The current opinion is that the efficiency of good implementations of simplex-based methods and interior point methods are similar for routine applications of linear programming.[7]

LP solvers are in widespread use for optimization of various problems in industry, such as optimization of flow in transportation networks.[8]

Open problems and recent work

There are several open problems in the theory of linear programming, the solution of which would represent fundamental breakthroughs in mathematics and potentially major advances in our ability to solve large-scale linear programs.

- Does LP admit a strongly polynomial-time algorithm?
- Does LP admit a strongly polynomial algorithm to find a strictly complementary solution?
- Does LP admit a polynomial algorithm in the real number (unit cost) model of computation?

This closely related set of problems has been cited by Stephen Smale as among the 18 greatest unsolved problems of the 21st century. In Smale's words, the third version of the problem "is the main unsolved problem of linear programming theory." While algorithms exist to solve linear programming in weakly polynomial time, such as the ellipsoid methods and interior-point techniques, no algorithms have yet been found that allow strongly

polynomial-time performance in the number of constraints and the number of variables. The development of such algorithms would be of great theoretical interest, and perhaps allow practical gains in solving large LPs as well.

- Are there pivot rules which lead to polynomial-time Simplex variants?
- Do all polytopal graphs have polynomially-bounded diameter?
- Is the Hirsch conjecture true for polytopal graphs?

These questions relate to the performance analysis and development of Simplex-like methods. The immense efficiency of the Simplex algorithm in practice despite its exponential-time theoretical performance hints that there may be variations of Simplex that run in polynomial or even strongly polynomial time. It would be of great practical and theoretical significance to know whether any such variants exist, particularly as an approach to deciding if LP can be solved in strongly polynomial time.

The Simplex algorithm and its variants fall in the family of edge-following algorithms, so named because they solve linear programming problems by moving from vertex to vertex along edges of a polytope. This means that their theoretical performance is limited by the maximum number of edges between any two vertices on the LP polytope. As a result, we are interested in knowing the maximum graph-theoretical diameter of polytopal graphs. It has been proved that all polytopes have subexponential diameter, and all experimentally observed polytopes have linear diameter, it is presently unknown whether any polytope has superpolynomial or even superlinear diameter. If any such polytopes exist, then no edge-following variant can run in polynomial or linear time, respectively. Questions about polytope diameter are of independent mathematical interest.

Simplex pivot methods preserve primal (or dual) feasibility. On the other hand, criss-cross pivot methods do not preserve (primal or dual) feasibility — they may visit primal feasible, dual feasible or primal-and-dual infeasible bases in any order. Pivot methods of this type have been studied since the 1970s. Essentially, these methods attempt to find the shortest pivot path on the arrangement polytope under the linear programming problem. In contrast to polytopal graphs, graphs of arrangement polytopes are known to have small diameter, allowing the possibility of strongly polynomial-time criss-cross pivot algorithm without resolving questions about the diameter of general polytopes.[9]

Integer unknowns

If the unknown variables are all required to be integers, then the problem is called an integer programming (IP) or **integer linear programming** (ILP) problem. In contrast to linear programming, which can be solved efficiently in the worst case, integer programming problems are in many practical situations (those with bounded variables) NP-hard. **0-1 integer programming** or **binary integer programming** (BIP) is the special case of integer programming where variables are required to be 0 or 1 (rather than arbitrary integers). This problem is also classified as NP-hard, and in fact the decision version was one of Karp's 21 NP-complete problems.

If only some of the unknown variables are required to be integers, then the problem is called a **mixed integer programming** (MIP) problem. These are generally also NP-hard.

There are however some important subclasses of IP and MIP problems that are efficiently solvable, most notably problems where the constraint matrix is totally unimodular and the right-hand sides of the constraints are integers.

Advanced algorithms for solving integer linear programs include:

- cutting-plane method
- branch and bound
- branch and cut
- branch and price
- if the problem has some extra structure, it may be possible to apply delayed column generation.

Such integer-programming algorithms are discussed by Padberg and in Beasley.

Solvers and scripting (programming) languages

Free, opensource:

Name	License	Brief info
LP_Solve [10]	LGPL	User-friendly linear and integer programming solver. Also provides DLL for program integration.
Cassowary constraint solver	LGPL	an incremental constraint solving toolkit that efficiently solves systems of linear equalities and inequalities.
Coopr [11]	BSD	Python packages for modeling and solving optimization problems
CVXOPT [12]	GPL	general purpose convex optimization solver written in Python, with a C API, and calls external routines (e.g. BLAS, LAPACK, FFTW) for numerical computations. Has its own solvers, but can also call glpk or MOSEK[13] if installed
glpk	GPL	GNU Linear Programming Kit, a free LP/MILP solver. Uses GNU MathProg modelling language.
OpenOpt	BSD	Universal cross-platform numerical optimization framework; see its LP [14] page and other problems [15] involved
pulp-or [16]	BSD	Python module for modeling and solving linear programming problems
Pyomo [17]	BSD	Python module for formulating linear programming problems with abstract models
Qoca	GPL	a library for incrementally solving systems of linear equations with various goal functions
CLP	CPL	an LP solver from COIN-OR project
R-Project	GPL	a programming language and software environment for statistical computing and graphics
CVX [18]	GPL	MATLAB based modeling system for convex optimization, including linear programs; calls either SDPT3 or SeDuMi as a solver
CVXMOD [19]	GPL	Python based modeling system, similar to CVX. It calls CVXOPT as its solver. It is still in in alpha release, as of 2009
SDPT3 [20]	GPL	MATLAB based convex optimization solver
SeDuMi [21]	GPL	MATLAB based convex optimization solver

MINTO, here [22] Mixed Integer Optimizer (an integer programming solver which uses branch and bound algorithm) has publicly available source code but not open source.

Proprietary:

Name	Brief info
APMonitor	
AIMMS	
AMPL	
CPLEX	Popular solver with an API for several programming languages, and also has a modelling language and works with TOMLAB
EXCEL Solver Function	
FortMP	
GAMS	
GIPALS	
Gurobi	
IMSL Numerical Libraries	Collections of math and statistical algorithms available in C/C++, Fortran, Java and C#/.NET. Optimization routines in the IMSL Libraries include unconstrained, linearly and nonlinearly constrained minimizations, and linear programming algorithms.

Lingo	
LPL	
MATLAB	A general-purpose and matrix-oriented programming-language for numerical computing. Linear programming in MATLAB requires the Optimization Toolbox in addition to the base MATLAB product; available routines include BINTPROG and LINPROG
Mathematica	A general-purpose programming-language for mathematics, including symbolic and numerical capabilities.
MOPS	
MOSEK	A solver for large scale optimization with API for several languages (C++,java,.net, Matlab and python).
NMath Stats	A general-purpose .NET statistical library containing a simplex solver.[23]
OptimJ	
SAS	
Solver Foundation	A .NET platform for modeling, scheduling, and optimization.
TOMLAB	
VisSim	A visual block diagram language for simulation of dynamical systems.
Xpress	

See also

- Mathematical programming
- Nonlinear programming
- Convex programming
- Simplex algorithm, used to solve LP problems
- Quadratic programming, a superset of linear programming
- Leonid Kantorovich, one of the founders of linear programming
- Shadow price
- MPS file format
- nl file format
- MIP example, job shop problem
- Linear-fractional programming (LFP)
- Oriented matroid
- *see also the "External links" section below*

Further reading

A reader may consider beginning with Nering and Tucker, with the first volume of Dantzig and Thapa, or with Williams.

- Dmitris Alevras and Manfred W. Padberg, *Linear Optimization and Extensions: Problems and Extensions*, Universitext, Springer-Verlag, 2001. (Problems from Padberg with solutions.)
- A. Bachem and W. Kern. *Linear Programming Duality: An Introdution to Oriented Matroids*. Universitext. Springer-Verlag, 1992. (Combinatorial)
- J. E. Beasley, editor. *Advances in Linear and Integer Programing*. Oxford Science, 1996. (Collection of surveys)
- Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Schwarzkopf (2000). *Computational Geometry* (2nd revised ed.). Springer-Verlag. ISBN 3-540-65620-0. Chapter 4: Linear Programming: pp. 63–94. Describes a randomized half-plane intersection algorithm for linear programming.
- R.G. Bland, New finite pivoting rules for the simplex method, *Math. Oper. Res.* 2 (1977) 103–107.

- Karl-Heinz Borgwardt, *The Simplex Algorithm: A Probabilistic Analysis*, Algorithms and Combinatorics, Volume 1, Springer-Verlag, 1987. (Average behavior on random problems)
- V. Chandru and M.R.Rao, Linear Programming, Chapter 31 in *Algorithms and Theory of Computation Handbook*, edited by M.J.Atallah, CRC Press 1999, 31-1 to 31-37.
- V. Chandru and M.R.Rao, Integer Programming, Chapter 32 in *Algorithms and Theory of Computation Handbook*, edited by M.J.Atallah, CRC Press 1999, 32-1 to 32-45.[24]
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Chapter 29: Linear Programming, pp. 770–821. (computer science)
- Richard W. Cottle, ed. *The Basic George B. Dantzig*. Stanford Business Books, Stanford University Press, Stanford, California, 2003. (Selected papers by George B. Dantzig)
- George B. Dantzig and Mukund N. Thapa. 1997. *Linear programming 1: Introduction*. Springer-Verlag.
- George B. Dantzig and Mukund N. Thapa. 2003. *Linear Programming 2: Theory and Extensions*. Springer-Verlag. (Comprehensive, covering e.g. pivoting and interior-point algorithms, large-scale problems, decomposition following Dantzig-Wolfe and Benders, and introducing stochastic programming.)
- Komei Fukuda and Tamás Terlaky, Criss-cross methods: A fresh view on pivot algorithms. (1997) *Mathematical Programming* Series B, Vol 79, Nos. 1—3, 369—395. (Invited survey, from the International Symposium on Mathematical Programming.)
- Michael R. Garey and David S. Johnson (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman. ISBN 0-7167-1045-5. A6: MP1: INTEGER PROGRAMMING, pg.245. (computer science, complexity theory)
- Bernd Gärtner, Jiří Matoušek (2006). *Understanding and Using Linear Programming*, Berlin: Springer. ISBN 3-540-30697-8 (introduction for mathematicians and computer scientists)
- Kattta G. Murty, *Linear Programming*, Wiley, 1983. (comprehensive reference to classical approaches)
- Evar D. Nering and Albert W. Tucker, 1993, *Linear Programs and Related Problems*, Academic Press. (elementary)
- M. Padberg, *Linear Optimization and Extensions*, Second Edition, Springer-Verlag, 1999. (carefully written account of primal and dual simplex algorithms and projective algorithms, with an introduction to integer linear programming --- featuring the traveling salesman problem for Odysseus.)
- Christos H. Papadimitriou and Kenneth Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Corrected republication with a new preface, Dover. (computer science)
- Cornelis Roos, Tamás Terlaky, Jean-Philippe Vial, *Interior Point Methods for Linear Optimization*, Second Edition, Springer-Verlag, 2006. (Graduate level)
- Alexander Schrijver, *Theory of Linear and Integer Programming*. John Wiley & sons, 1998, ISBN 0-471-98232-6 (mathematical)
- Michael J. Todd (February 2002). "The many facets of linear programming". *Mathematical Programming* **91** (3). (Invited survey, from the International Symposium on Mathematical Programming.)
- Robert J. Vanderbei, *Linear Programming: Foundations and Extensions* [25], 3rd ed., International Series in Operations Research & Management Science, Vol. 114, Springer Verlag, 2008. ISBN 978-0-387-74387-5. (An on-line second edition was formerly available. Vanderbei's site still contains extensive materials.)
- Vazirani, Vijay V. (2001). *Approximation Algorithms*. Springer-Verlag. ISBN 3-540-65367-8. (Computer science)
- H. P. Williams, *Model Buildiing in Mathematical Programming*, Third revised Edition, 1990. (Modeling)
- Stephen J. Wright, 1997, *Primal-Dual Interior-Point Methods*, SIAM. (Graduate level)
- Yinyu Ye, 1997, *Interior Point Algorithms: Theory and Analysis*, Wiley. (Advanced graduate-level)
- Ziegler, Günter M., Chapters 1-3 and 6-7 in *Lectures on Polytopes*, Springer-Verlag, New York, 1994. (Geometry)

External links

- Guidance on Formulating LP problems [26]
- 0-1 Integer Programming Benchmarks with Hidden Optimum Solutions [27]
- Mathematical Programming Glossary [2]
- A Tutorial on Integer Programming [28]
- The linear programming FAQ [29]
- Linear Programming Survey OR/MS Today [30]
- Linear Programming: Guide to Formulation, Simplex Algorithm, Goal Programming and Excel Solver examples [31]
- George Dantzig [32]

Time complexity

In computer science, the **time complexity** of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the size of the input to the problem. The time complexity of an algorithm is commonly expressed using the big O notation, which suppresses multiplicative constants and lower order terms. When expressed this way, the time complexity is said to be described *asymptotically*, i.e., as the input size goes to infinity. For example, if the time required by an algorithm on all inputs of size n is at most $5n^3 + 3n$, the asymptotic time complexity is $O(n^3)$.

Time complexity is commonly estimated by counting the number of elementary operations performed by the algorithm, where an elementary operation takes a fixed amount of time to perform. Thus the amount of time taken and the number of elementary operations performed by the algorithm differ by at most a constant factor.

Since an algorithm may take a different amount of time even on inputs of the same size, the most commonly used measure of time complexity, the worst-case time complexity of an algorithm, denoted as $\boldsymbol{T(n)}$, is the maximum amount of time taken on any input of size n. Time complexities are classified by the nature of the function $T(n)$. For instance, an algorithm with $T(n) = O(n)$ is called a linear time algorithm, and an algorithm with $T(n) = O(2^n)$ is said to be an exponential time algorithm.

Table of common time complexities

The following table summarises some classes of commonly encountered time complexities. In the table, $\text{poly}(x) = x^{O(1)}$, i.e., polynomial in x.

Name	Complexity class	Running time ($T(n)$)	Examples of running times	Examples of algorithms
constant time		$O(1)$	10	Determining if a number is even or odd
inverse Ackermann		$O(\alpha(n))$		Amortized time per operation using a disjoint set
iterated logarithmic		$O(\log^* n)$		Distributed coloring of cycles
log-logarithmic		$O(\log \log n)$		Amortized time per operation using a bounded priority queue[1]
logarithmic time	DLOGTIME	$O(\log n)$	$\log n$, $\log(n^2)$	Binary search
polylogarithmic time		$\text{poly}(\log n)$	$(\log n)^2$	
fractional power		$O(n^c)$, $0<c<1$	$n^{1/2}$, $n^{2/3}$	Searching in a kd-tree
linear time		$O(n)$	n	Finding the smallest item in an unsorted array

"n log star n"		$O(n \log^* n)$		Seidel's polygon triangulation algorithm. "log star n" is the iterated logarithm
linearithmic time		$O(n \log n)$	$n \log n$, $\log n!$	Fastest possible comparison sort
quadratic time		$O(n^2)$	n^2	Bubble sort; Insertion sort
cubic time		$O(n^3)$	n^3	Naive multiplication of two $n \times n$ matrices. Calculating partial correlation.
polynomial time	P	$2^{O(\log n)} = \text{poly}(n)$	n, $n \log n$, n^{10}	Karmarkar's algorithm for linear programming; AKS primality test
quasi-polynomial time	QP	$2^{\text{poly}(\log n)}$	$n^{\log \log n}$, $n^{\log n}$	Best-known $O(\log^2 n)$-approximation algorithm for the directed Steiner tree problem.
sub-exponential time (first definition)	SUBEXP	$O(2^{n^\varepsilon})$ for all $\varepsilon > 0$	$O(2^{\log n} \log \log n)$	Assuming complexity theoretic conjectures, BPP is contained in SUBEXP.[2]
sub-exponential time (second definition)		$2^{o(n)}$	$2^n 1/3$	Best-known algorithm for integer factorization and graph isomorphism
exponential time	E	$2^{O(n)}$	1.1^n, 10^n	Solving the traveling salesman problem using dynamic programming
exponential time	EXPTIME	$2^{\text{poly}(n)}$	$n!$, n^n, $2^n 2$	
factorial time		$O(n!)$	$n!$	Solving the traveling salesman problem via brute-force search
double exponential time	2-EXPTIME	$2^{2\text{poly}(n)}$	$2^3 n$	Deciding the truth of a given statement in Presburger arithmetic

Constant time

An algorithm is said to be **constant time** (also written as **O(1)** time) if the value of $T(n)$ is bounded by a value that does not depend on the size of the input. For example, accessing any single element in an array takes constant time as only one operation has to be performed to locate it. However, finding the minimal value in an unordered array is not a constant time operation as a scan over each element in the array is needed in order to determine the minimal value. Hence it is a linear time operation, taking O(n) time (unless some more efficient algorithm is devised, for example, a binary search across a sorted array). If the number of elements is known in advance and does not change, however, such an algorithm can still be said to run in constant time.

Despite the name "constant time", the running time does not have to be independent of the problem size, but an upper bound for the running time has to be bounded independently of the problem size. For example, the task "exchange the values of a and b if necessary so that $a \leq b$" is called constant time even though the time may depend on whether or not it is already true that $a \leq b$. However, there is some constant t such that the time required is always *at most* t.

Here are some examples of code fragments that run in constant time:

```
int index = 5;
int item = list[index];
if (condition true) then
   perform some operation that runs in constant time
else
   perform some other operation that runs in constant time
for i = 1 to 100
   for j = 1 to 200
      perform some operation that runs in constant time
```

If $T(n)$ is O(*any constant value*), this is equivalent to and stated in standard notation as $T(n)$ being O(1).

Logarithmic time

An algorithm is said to take **logarithmic time** if $T(n) = \mathbf{O(\log \boldsymbol{n})}$. Due to the use of the binary numeral system by computers, the logarithm is frequently base 2 (that is, $\log_2 n$, sometimes written lg n). However, by the change of base equation for logarithms, $\log_a n$ and $\log_b n$ differ only by a constant multiplier, which in big-O notation is discarded; thus O(log n) is the standard notation for logarithmic time algorithms regardless of the base of the logarithm.

Algorithms taking logarithmic time are commonly found in operations on binary trees.

Polylogarithmic time

An algorithm is said to run in **polylogarithmic time** if $T(n) = \mathrm{O}((\log n)^k)$, for some constant k. For example, matrix chain ordering can be solved in polylogarithmic time on a Parallel Random Access Machine.[3]

Sub-linear time

An algorithm is said to run in **sub-linear time** (often spelled **sublinear time**) if $T(n) = \mathrm{o}(n)$. In particular this includes algorithms with the time complexities defined above, as well as others such as the $\mathrm{O}(n^{1/2})$ Grover's search algorithm.

For an algorithm to be exact and yet run in sub-linear time, it needs to use parallel processing (as the NC_1 matrix determinant calculation does) or non-classical processing (as Grover's search does), or alternatively have guaranteed assumptions on the input structure (as the logarithmic time binary search and many tree maintenance algorithms do). Otherwise, a sub-linear time algorithm would not be able to read or learn the entire input prior to providing its output.

The specific term *sublinear time algorithm* is usually reserved to algorithms that are unlike the above in that they are run over classical serial machine models and are not allowed prior assumptions on the input[4] . They are however allowed to be randomized, and indeed must be randomized for all but the most trivial of tasks.

As such an algorithm must provide an answer without reading the entire input, its particulars heavily depend on the access allowed to the input. Usually for an input that is represented as a binary string $b_1,...,b_k$ it is assumed that the algorithm can in time O(1) request and obtain the value of b_i for any i.

Sublinear time algorithms are typically randomized, and provide only approximate solutions. In fact, the property of a binary string having only zeros (and no ones) can be easily proved not to be decidable by a (non-approximate) sublinear time algorithm. Sublinear time algorithms arise naturally in the investigation of property testing.

Linear time

An algorithm is said to take **linear time**, or $\mathbf{O(\boldsymbol{n})}$ time, if its time complexity is O(n). Informally, this means that for large enough input sizes the running time increases linearly with the size of the input. For example, a procedure that adds up all elements of a list requires time proportional to the length of the list. This description is slightly inaccurate, since the running time can significantly deviate from a precise proportionality, especially for small values of n.

Linear time is often viewed as a desirable attribute for an algorithm. Much research has been invested into creating algorithms exhibiting (nearly) linear time or better. This research includes both software and hardware methods. In the case of hardware, some algorithms which, mathematically speaking, can never achieve linear time with standard computation models are able to run in linear time. There are several hardware technologies which exploit parallelism to provide this. An example is content-addressable memory. This concept of Linear Time is used in string matching

algorithms such as Boyer-Moore Algorithm and Ukkonen's Algorithm.

Linearithmic/quasilinear time

A **linearithmic function** (portmanteau of *linear* and *logarithmic*) is a function of the form $n \cdot \log n$ (i.e., a product of a linear and a logarithmic term). An algorithm is said to run in **linearithmic time** (also known as **quasilinear time**) if $T(n) = \mathbf{O(}n \log n\mathbf{)}$. Compared to other functions, a linearithmic function is $\omega(n)$, $o(n^{1+\varepsilon})$ for every $\varepsilon > 0$, and $\Theta(n \cdot \log n)$. Thus, a linearithmic term grows faster than a linear term but slower than any polynomial in n with exponent strictly greater than 1.

In many cases, the $n \cdot \log n$ running time is simply the result of performing a $\Theta(\log n)$ operation n times. For example, Binary tree sort creates a Binary tree by inserting each element of the n-sized array one by one. Since the insert operation on a self-balancing binary search tree takes $O(\log n)$ time, the entire algorithm takes linearithmic time.

Comparison sorts require at least linearithmic number of comparisons in the worst case because $\log(n!) = \Theta(n \log n)$. They also frequently arise from the recurrence relation $T(n) = 2\,T(n/2) + O(n)$.

Some famous algorithms that run in linearithmic time include:

- Comb sort, in the average and worst case
- Quicksort in the average case
- Heapsort, merge sort, introsort, binary tree sort, smoothsort, comb sort, patience sorting, etc. in the worst case
- Fast Fourier transforms
- Monge array calculation

Sub-quadratic time

An algorithm is said to be **subquadratic time** if $T(n) = o(n^2)$.

For example, most naïve comparison-based sorting algorithms are quadratic (e.g. insertion sort), but more advanced algorithms can be found that are subquadratic (e.g. merge sort); to be precise, such algorithms are linearithmic. No general-purpose sorts run in linear time, but the change from quadratic to the common $O(n \log n)$ is of great practical importance.

Polynomial time

An algorithm is said to be **polynomial time** if its running time is upper bounded by a polynomial in the size of the input for the algorithm, i.e., $T(n) = O(n^k)$ for some constant k.[5] [6] Problems for which a polynomial time algorithm exists belong to the complexity class **P**, which is central in the field of computational complexity theory. Cobham's thesis states that polynomial time is a synonym for "tractable", "feasible", "efficient", or "fast".[7]

Some examples of polynomial time algorithms:

- The quicksort sorting algorithm on n integers performs at most An^2 operations for some constant A. Thus it runs in time $O(n^2)$ and is a polynomial time algorithm.
- All the basic arithmetic operations (addition, subtraction, multiplication, division, and comparison) can be done in polynomial time.
- Maximum matchings in graphs can be found in polynomial time.

Strongly and weakly polynomial time

In some contexts, especially in optimization, one differentiates between **strongly polynomial time** and **weakly polynomial time** algorithms. These two concepts are only relevant if the inputs to the algorithms consist of integers.

Strongly polynomial time is defined in the arithmetic model of computation. In this model of computation the basic arithmetic operations (addition, subtraction, multiplication, division, and comparison) take a unit time step to perform, regardless of the sizes of the operands. The algorithm runs in strongly polynomial time if [8]

1. the number of operations in the arithmetic model of computation is bounded by a polynomial in the number of integers in the input instance; and
2. the space used by the algorithm is bounded by a polynomial in the size of the input.

Any algorithm with these two properties can be converted to a polynomial time algorithm by replacing the arithmetic operations by suitable algorithms for performing the arithmetic operations on a Turing machine. If the second requirement above is omitted, then this is not true any more. Given n integers it is possible to compute 2^{2^n} with n multiplications using repeated squaring. If the integers are small enough (say they are equal to 1), then 2^{2^n} cannot be represented in polynomial space. Hence, it is not possible to compute this number in polynomial time on a Turing machine, but it is possible to compute it by polynomially many arithmetic operations.

There are algorithms which run in polynomial time in the Turing machine model but not in the arithmetic model. The Euclidean algorithm for computing the greatest common divisor of two integers is one example. Given two integers a and b the running time of the algorithm is bounded by $O((\log\ a + \log\ b)^2)$. This is polynomial in the size of a binary representation of a and b as the size of such a representation is roughly $\log\ a + \log\ b$. However, the algorithm does not run in strongly polynomial time as the running time depends on the magnitudes of a and b and not only on the number of integers in the input (which is constant in this case, there is always only two integers in the input).

An algorithm which runs in polynomial time but which is not strongly polynomial is said to run in **weakly polynomial time**.[9] A well-known example of a problem for which a weakly polynomial-time algorithm is known, but is not known to admit a strongly polynomial-time algorithm, is linear programming. Weakly polynomial-time should not be confused with pseudo-polynomial time.

Complexity classes

The concept of polynomial time leads to several complexity classes in computational complexity theory. Some important classes defined using polynomial time are the following.

- **P**: The complexity class of decision problems that can be solved on a deterministic Turing machine in polynomial time.
- **NP**: The complexity class of decision problems that can be solved on a non-deterministic Turing machine in polynomial time.
- **ZPP**: The complexity class of decision problems that can be solved with zero error on a probabilistic Turing machine in polynomial time.
- **RP**: The complexity class of decision problems that can be solved with 1-sided error on a probabilistic Turing machine in polynomial time.
- **BPP**: The complexity class of decision problems that can be solved with 2-sided error on a probabilistic Turing machine in polynomial time.
- **BQP**: The complexity class of decision problems that can be solved with 2-sided error on a quantum Turing machine in polynomial time.

P is the smallest time-complexity class on a deterministic machine which is robust in terms of machine model changes. (For example, a change from a single-tape Turing machine to a multi-tape machine can lead to a quadratic speedup, but any algorithm that runs in polynomial time under one model also does so on the other.) Any given

abstract machine will have a complexity class corresponding to the problems which can be solved in polynomial time on that machine.

Superpolynomial time

An algorithm is said to take **superpolynomial time** if $T(n)$ is not bounded above by any polynomial. It is $\omega(n^c)$ time for all constants c, where n is the input parameter, typically the number of bits in the input.

For example, an algorithm that runs for 2^n steps on an input of size n requires superpolynomial time (more specifically, exponential time).

An algorithm that uses exponential resources is clearly superpolynomial, but some algorithms are only very weakly superpolynomial. For example, the Adleman–Pomerance–Rumely primality test runs for $n^{O(\log \log n)}$ time on n-bit inputs; this grows faster than any polynomial for large enough n, but the input size must become impractically large before it cannot be dominated by a polynomial with small degree.

An algorithm that has been proven to require superpolynomial time cannot be solved in polynomial time, and so is known to lie outside the complexity class **P**. Cobham's thesis conjectures that these algorithms are impractical, and in many cases they are. Since the P versus NP problem is unresolved, no algorithm for a NP-complete problem is currently known to run in polynomial time.

Quasi-polynomial time

Quasi-polynomial time algorithms are algorithms which run slower than polynomial time, yet not so slow as to be exponential time. The worst case running time of a quasi-polynomial time algorithm is $2^{O((\log n)^c)}$for some fixed c . The best-known classical algorithm for integer factorization, the general number field sieve, which runs in time about $2^{\tilde{O}(n^{1/3})}$is *not* quasi-polynomial since the running time cannot be expressed as $2^{O((\log n)^c)}$for some fixed c . If the constant "c" in the definition of quasi-polynomial time algorithms is equal to 1, we get a polynomial time algorithm, and if it less than 1, we get a sub-linear time algorithm.

Quasi-polynomial time algorithms typically arise in reductions from an NP-hard problem to another problem. For example, one can take an instance of an NP hard problem, say 3SAT, and convert it to an instance of another problem B, but the size of the instance becomes $2^{O((\log n)^c)}$. In that case, this reduction does not prove that problem B is NP-hard; this reduction only shows that there is no polynomial time algorithm for B unless there is a quasi-polynomial time algorithm for 3SAT (and thus all of NP). Similarly, there are some problems for which we know quasi-polynomial time algorithms, but no polynomial time algorithm is known. Such problems arise in approximation algorithms; a famous example is the directed Steiner tree problem problem, for which there is a quasi-polynomial time approximation algorithm achieving an approximation factor of $O(\log^2 n)$(n being the number of vertices), but showing the existence of such a polynomial time algorithm is an open problem.

The complexity class **QP** consists of all problems which have quasi-polynomial time algorithms. It can be defined in terms of DTIME as follows.[10]

$$\mathrm{QP} = \bigcup_{c \in \mathbb{N}} \mathrm{DTIME}(2^{(\log n)^c})$$

Relation to NP-complete problems

In complexity theory, the P versus NP problem asks if all problems in NP have polynomial-time algorithms. All the best-known algorithms for NP-complete problems like 3SAT etc. take exponential time. Indeed, it is conjectured for many natural NP-complete problems that they do not have sub-exponential time algorithms. Here "sub-exponential time" is taken to mean the second definition presented above. (On the other hand, many graph problems represented in the natural way by adjacency matrices are solvable in subexponential time simply because the size of the input is square of the number of vertices.) This conjecture (for the k-SAT problem) is known as the exponential time

hypothesis.[11] Since it is conjectured that NP-complete problems do not have quasi-polynomial time algorithms, some inapproximability results in the field of approximation algorithms make the assumption that NP-complete problems do not have quasi-polynomial time algorithms. For example, see the known inapproximability results for the set cover problem.

Sub-exponential time

The term **sub-exponential time** is used to express that the running time of some algorithm may grow faster than any polynomial but is still significantly smaller than an exponential. In this sense, problems that have sub-exponential time algorithms are somewhat more tractable than those that only have exponential algorithms. The precise definition of "sub-exponential" is not generally agreed upon,[12] and we list the two most widely-used ones below.

First definition

A problem is said to be sub-exponential time solvable if it can be solved in running times whose logarithms grow smaller than any given polynomial. More precisely, a problem is in sub-exponential time if for every $\varepsilon > 0$ there exists an algorithm which solves the problem in time $O(2^{n\varepsilon})$. The set of all such problems is the complexity class **SUBEXP** which can be defined in terms of DTIME as follows.[2] [13] [14] [15]

$$\mathrm{SUBEXP} = \bigcap_{\varepsilon>0} \mathrm{DTIME}\left(2^{n^{\varepsilon}}\right)$$

Note that this notion of sub-exponential is non-uniform in terms of ε in the sense that ε is not part of the input and each ε may have its own algorithm for the problem.

Second definition

Some authors define sub-exponential time as running times in $2^{o(n)}$.[11] [16] [17] This definition allows larger running times than the first definition of sub-exponential time. An example of such a sub-exponential time algorithm is the best-known classical algorithm for integer factorization, the general number field sieve, which runs in time about $2^{\tilde{O}(n^{1/3})}$, where the length of the input is *n*. Another example is the best-known algorithm for the graph isomorphism problem, which runs in time $2^{O(\sqrt{(n \log n)})}$.

Note that it makes a difference whether the algorithm is allowed to be sub-exponential in the size of the instance, the number of vertices, or the number of edges. In parameterized complexity, this difference is made explicit by considering pairs (L, k) of decision problems and parameters k. **SUBEPT** is the class of all parameterized problems that run in time sub-exponential in *k* and polynomial in the input size *n*:[18]

$$\mathrm{SUBEPT} = \mathrm{DTIME}\left(2^{o(k)} \cdot \mathrm{poly}(n)\right).$$

More precisely, SUBEPT is the class of all parameterized problems (L, k) for which there is a computable function $f : \mathbb{N} \to \mathbb{N}$ with $f \in o(k)$ and an algorithm that decides L in time $2^{f(k)} \cdot \mathrm{poly}(n)$.

Exponential time hypothesis

The **exponential time hypothesis** (**ETH**) is that 3SAT, the satisfiability problem of Boolean formulas in conjunctive normal form with at most three literals per clause and with n variables, cannot be solved in time $2^{o(n)}$. With m denoting the number of clauses, ETH is equivalent to the hypothesis that *k*SAT cannot be solved in time $2^{o(m)}$ for any integer $k \geq 3$.[19] The exponential time hypothesis implies P ≠ NP.

Exponential time

An algorithm is said to be **exponential time**, if $T(n)$ is upper bounded by $2^{\text{poly}(n)}$, where poly(n) is some polynomial in n. More formally, an algorithm is exponential time if $T(n)$ is bounded by $O(2^{n^k})$ for some constant k. Problems which admit exponential time algorithms on a deterministic Turing machine form the complexity class known as **EXP**.

$$\mathrm{EXP} = \bigcup_{c \in \mathbb{N}} \mathrm{DTIME}\left(2^{n^c}\right)$$

Sometimes, exponential time is used to refer to algorithms that have $T(n) = 2^{O(n)}$, where the exponent is at most a linear function of n. This gives rise to the complexity class **E**.

$$\mathrm{E} = \bigcup_{c \in \mathbb{N}} \mathrm{DTIME}\left(2^{cn}\right)$$

Double exponential time

An algorithm is said to be double exponential time if $T(n)$ is upper bounded by $2^{2^{\text{poly}(n)}}$, where poly(n) is some polynomial in n. Such algorithms belong to the complexity class 2-EXPTIME.

$$\text{2-EXPTIME} = \bigcup_{c \in \mathbb{N}} \mathrm{DTIME}(2^{2^{n^c}})$$

Well-known double exponential time algorithms include:

- Decision procedures for Presburger arithmetic
- Computing a Gröbner basis (in the worst case)
- Finding a complete set of associative-commutative unifiers[20]
- Satisfying CTL^+ (which is, in fact, 2-EXPTIME-complete)[21]
- Quantifier elimination on real closed fields takes at least doubly-exponential time (but is not even known to be computable in ELEMENTARY)

See also

- L-notation

References

[1] Mehlhorn, Kurt; Naher, Stefan (1990). "Bounded ordered dictionaries in O(log log N) time and O(n) space". *Information Processing Letters*.

[2] Babai, László; Fortnow, Lance; Nisan, N.; Wigderson, Avi (1993). "BPP has subexponential time simulations unless EXPTIME has publishable proofs". *Computational Complexity* (Berlin, New York: Springer-Verlag) **3** (4): 307–318. doi:10.1007/BF01275486.

[3] Bradford, Phillip G.; Rawlins, Gregory J. E.; Shannon, Gregory E. (1998). "Efficient Matrix Chain Ordering in Polylog Time". *SIAM Journal on Computing* (Philadelphia: Society for Industrial and Applied Mathematics) **27** (2): 466–490. doi:10.1137/S0097539794270698. ISSN 1095-7111.

[4] Kumar, Ravi; Rubinfeld, Ronitt (2003). "Sublinear time algorithms" (http://www.cs.princeton.edu/courses/archive/spr04/cos598B/bib/kumarR-survey.pdf). *SIGACT News* **34** (4): 57–67. ..

[5] Papadimitriou, Christos H. (1994). *Computational complexity*. Reading, Mass.: Addison-Wesley. ISBN 0-201-53082-1.

[6] Sipser, Michael (2006). *Introduction to the Theory of Computation*. Course Technology Inc. ISBN 0-619-21764-2.

[7] Cobham, Alan (1965). "The intrinsic computational difficulty of functions". *Proc. Logic, Methodology, and Philosophy of Science II*. North Holland.
[8] Grötschel, Martin; László Lovász, Alexander Schrijver (1988). "Complexity, Oracles, and Numerical Computation". *Geometric Algorithms and Combinatorial Optimization*. Springer. ISBN 038713624X.
[9] Schrijver, Alexander (2003). "Preliminaries on algorithms and Complexity". *Combinatorial Optimization: Polyhedra and Efficiency*. **1**. Springer. ISBN 3540443894.
[10] *Complexity Zoo*: Class QP: Quasipolynomial-Time (http://qwiki.stanford.edu/wiki/Complexity_Zoo:Q#qp)
[11] Impagliazzo, R.; Paturi, R. (2001). "On the complexity of k-SAT". *Journal of Computer and System Sciences* (Elsevier) **62** (2): 367–375. doi:10.1006/jcss.2000.1727. ISSN 1090-2724.
[12] Aaronson, Scott (5 April 2009). "A not-quite-exponential dilemma" (http://scottaaronson.com/blog/?p=394). *Shtetl-Optimized*. . Retrieved 2 December 2009.
[13] *Complexity Zoo*: Class SUBEXP: Deterministic Subexponential-Time (http://qwiki.stanford.edu/wiki/Complexity_Zoo:S#subexp)
[14] Moser, P. (2003). "Baire's Categories on Small Complexity Classes". *Lecture Notes in Computer Science* (Berlin, New York: Springer-Verlag): 333–342. ISSN 0302-9743.
[15] Miltersen, P.B. (2001). "DERANDOMIZING COMPLEXITY CLASSES". *Handbook of Randomized Computing* (Kluwer Academic Pub): 843.
[16] Kuperberg, Greg (2005). "A Subexponential-Time Quantum Algorithm for the Dihedral Hidden Subgroup Problem". *SIAM Journal on Computing* (Philadelphia: Society for Industrial and Applied Mathematics) **35** (1): 188. ISSN 1095-7111.
[17] Oded Regev (2004). "A Subexponential Time Algorithm for the Dihedral Hidden Subgroup Problem with Polynomial Space". *arXiv:quant-ph/0406151v1* [quant-ph].
[18] Flum, Jörg; Grohe, Martin (2006). *Parameterized Complexity Theory* (http://www.springer.com/east/home/generic/search/results?SGWID=5-40109-22-141358322-0). Springer. p. 417. ISBN 978-3-540-29952-3. . Retrieved 2010-03-05.
[19] Impagliazzo, R.; Paturi, R.; Zane, F. (2001). "Which problems have strongly exponential complexity?". *Journal of Computer and System Sciences* **63** (4): 512–530. doi:10.1006/jcss.2001.1774.
[20] Kapur, Deepak; Narendran, Paliath (1992). "Proc. 7th IEEE Symp. Logic in Computer Science (LICS 1992)" (http://citeseer.ist.psu.edu/337363.html). pp. 11–21. doi:10.1109/LICS.1992.185515. ..
[21] Johannse, Jan; Lange, Martin (2003). "CTL$^+$ is complete for double exponential time" (http://www.tcs.informatik.uni-muenchen.de/~mlange/papers/icalp03.pdf). in Baeten, Jos C. M.; Lenstra, Jan Karel; Parrow, Joachim et al.. *Proc. 30th Int. Colloq. Automata, Languages, and Programming (ICALP 2003)*. Lecture Notes in Computer Science. **2719**. Springer-Verlag. pp. 767–775. doi:10.1007/3-540-45061-0_60. ..

NP (complexity)

In computational complexity theory, **NP** is one of the most fundamental complexity classes. The abbreviation **NP** refers to "nondeterministic polynomial time".

Intuitively, **NP** is the set of all decision problems for which the 'yes'-answers have efficiently verifiable proofs of the fact that the answer is indeed 'yes'. More precisely, these proofs have to be *verifiable* in polynomial time by a deterministic Turing machine. In an equivalent formal definition, **NP** is the set of decision problems solvable in polynomial time by a non-deterministic Turing machine.

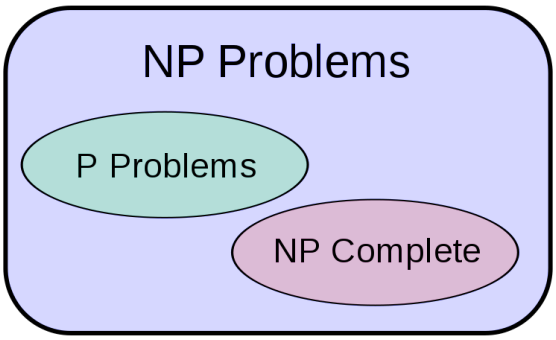


Diagram of complexity classes provided that **P** ≠ **NP**. The existence of problems outside both **P** and **NP**-complete in this case was established by Ladner.[1]

The complexity class **P** is contained in **NP**, but **NP** contains many important problems, the hardest of which are called **NP**-complete problems, for which no polynomial-time algorithms are known. The most important open question in complexity theory, the **P** = **NP** problem, asks whether such algorithms actually exist for **NP**-complete, and by corollary, all **NP** problems. It is widely believed that this is not the case.[2]

Formal definition

The complexity class **NP** can be defined in terms of NTIME as follows:

$$\mathrm{NP} = \bigcup_{k \in \mathbb{N}} \mathrm{NTIME}(n^k).$$

Alternatively, **NP** can be defined using deterministic Turing machines as verifiers. A language *L* is in **NP** if and only if there exist polynomials *p* and *q*, and a deterministic Turing machine *M*, such that

- For all *x* and *y*, the machine *M* runs in time *p*(|x|) on input *(x,y)*
- For all *x* in *L*, there exists a string *y* of length *q*(|x|) such that *M(x,y)* = 1
- For all *x* not in *L* and all strings *y* of length *q*(|x|), *M(x,y)* = 0

Introduction

Many natural computer science problems are covered by the class **NP**. In particular, the decision versions of many interesting search problems and optimization problems are contained in **NP**.

Verifier-based definition

In order to explain the verifier-based definition of **NP**, let us consider the subset sum problem: Assume that we are given some integers, such as {−7, −3, −2, 5, 8}, and we wish to know whether some of these integers sum up to zero. In this example, the answer is 'yes', since the subset of integers {−3, −2, 5} corresponds to the sum (−3) + (−2) + 5 = 0. The task of deciding whether such a subset with sum zero exists is called the *subset sum problem*.

As the number of integers that we feed into the algorithm becomes larger, the number of subsets grows exponentially, and in fact the subset sum problem is **NP**-complete. However, notice that, if we are given a particular subset (often called a *certificate*), we can easily check or *verify* whether the subset sum is zero, by just summing up the integers of the subset. So if the sum is indeed zero, that particular subset is the *proof* or witness for the fact that the answer is 'yes'. An algorithm that verifies whether a given subset has sum zero is called *verifier*. A problem is said to be in **NP** if and only if there exists a verifier for the problem that executes in polynomial time. In case of the subset sum problem, the verifier needs only polynomial time, for which reason the subset sum problem is in **NP**.

Note that the verifier-based definition of **NP** does *not* require an easy-to-verify certificate for the 'no'-answers. The class of problems with such certificates for the 'no'-answers is called **co-NP**. In fact, it is an open question whether all problems in **NP** also have certificates for the 'no'-answers and thus are in **co-NP**.

Machine-definition

Equivalent to the verifier-based definition is the following characterization: **NP** is the set of decision problems solvable in polynomial time by a non-deterministic Turing machine.

Examples

This is an incomplete list of problems that are in **NP**.

- All problems in **P** (For, given a certificate for a problem in **P**, we can ignore the certificate and just solve the problem in polynomial time. Alternatively, note that a deterministic Turing machine is also trivially a non-deterministic Turing machine that just happens to not use any non-determinism.)
- The decision problem version of the integer factorization problem: given integer *n* and *k*, is there a factor *f* with 1 < *f* < *k* and *f* dividing *n*?
- The graph isomorphism problem of determining whether two graphs can be drawn identically
- All NP-complete problems, e.g.:
  - A variant of the traveling salesman problem, where we want to know if there is a route of some length that goes through all the nodes in a certain network

- The boolean satisfiability problem, where we want to know if a certain formula in propositional logic with boolean variables can be true for some value of the variables or not

Why some NP problems are hard to solve

Because of the many important problems in this class, there have been extensive efforts to find polynomial-time algorithms for problems in **NP**. However, there remain a large number of problems in **NP** that defy such attempts, seeming to require super-polynomial time. Whether these problems really aren't decidable in polynomial time is one of the greatest open questions in computer science (see **P=NP problem** for an in-depth discussion).

An important notion in this context is the set of NP-complete decision problems, which is a subset of **NP** and might be informally described as the "hardest" problems in **NP**. If there is a polynomial-time algorithm for even *one* of them, then there is a polynomial-time algorithm for *all* the problems in **NP**. Because of this, and because dedicated research has failed to find a polynomial algorithm for any **NP-complete** problem, once a problem has been proven to be **NP-complete** this is widely regarded as a sign that a polynomial algorithm for this problem is unlikely to exist.

Equivalence of definitions

The two definitions of **NP** as the class of problems solvable by a nondeterministic Turing machine (TM) in polynomial time and the class of problems verifiable by a deterministic Turing machine in polynomial time are equivalent. The proof is described by many textbooks, for example Sipser's *Introduction to the Theory of Computation*, section 7.3.

To show this, first suppose we have a deterministic verifier. A nondeterministic machine can simply nondeterministically run the verifier on all possible proof strings (this requires only polynomially-many steps because it can nondeterministically choose the next character in the proof string in each step, and the length of the proof string must be polynomially bounded). If any proof is valid, some path will accept; if no proof is valid, the string is not in the language and it will reject.

Conversely, suppose we have a nondeterministic TM called A accepting a given language L. At each of its polynomially-many steps, the machine's computation tree branches in at most a constant number of directions. There must be at least one accepting path, and the string describing this path is the proof supplied to the verifier. The verifier can then deterministically simulate A, following only the accepting path, and verifying that its accepts at the end. If A rejects the input, there is no accepting path, and the verifier will never accept.

Relationship to other classes

NP contains all problems in **P**, since one can verify any instance of the problem by simply ignoring the proof and solving it. **NP** is contained in **PSPACE**—to show this, it suffices to construct a **PSPACE** machine that loops over all proof strings and feeds each one to a polynomial-time verifier. Since a polynomial-time machine can only read polynomially-many bits, it cannot use more than polynomial space, nor can it read a proof string occupying more than polynomial space (so we don't have to consider proofs longer than this). **NP** is also contained in EXPTIME, since the same algorithm operates in exponential time.

The complement of **NP**, co-NP, contains those problems which have a simple proof for *no* instances, sometimes called counterexamples. For example, primality testing trivially lies in co-NP, since one can refute the primality of an integer by merely supplying a nontrivial factor. **NP** and co-NP together form the first level in the polynomial hierarchy, higher only than **P**.

NP is defined using only deterministic machines. If we permit the verifier to be probabilistic (specifically, a BPP machine), we get the class **MA** solvable using a Arthur-Merlin protocol with no communication from Merlin to Arthur.

NP is a class of decision problems; the analogous class of function problems is FNP.

Other characterizations

There is also a simple logical characterization of **NP**: it contains precisely those languages expressible in second-order logic restricted to exclude universal quantification over relations, functions, and subsets.

NP can be seen as a very simple type of interactive proof system, where the prover comes up with the proof certificate and the verifier is a deterministic polynomial-time machine that checks it. It is complete because the right proof string will make it accept if there is one, and it is sound because the verifier cannot accept if there is no acceptable proof string.

A major result of complexity theory is that **NP** can be characterized as the problems solvable by probabilistically checkable proofs where the verifier uses O(log n) random bits and examines only a constant number of bits of the proof string (the class **PCP**(log n, 1)). More informally, this means that the **NP** verifier described above can be replaced with one that just "spot-checks" a few places in the proof string, and using a limited number of coin flips can determine the correct answer with high probability. This allows several results about the hardness of approximation algorithms to be proven.

Example

The decision version of the traveling salesman problem is in **NP**. Given an input matrix of distances between n cities, the problem is to determine if there is a route visiting all cities with total distance less than k.

A proof certificate can simply be a list of the cities. Then verification can clearly be done in polynomial time by a deterministic Turing machine. It simply adds the matrix entries corresponding to the paths between the cities.

A nondeterministic Turing machine can find such a route as follows:

- At each city it visits it "guesses" the next city to visit, until it has visited every vertex. If it gets stuck, it stops immediately.
- At the end it verifies that the route it has taken has cost less than k in O(n) time.

One can think of each guess as "forking" a new copy of the Turing machine to follow each of the possible paths forward, and if at least one machine finds a route of distance less than k, that machine accepts the input. (Equivalently, this can be thought of as a single Turing machine that always guesses correctly)

Binary search on the range of possible distances can convert the decision version of Traveling Salesman to the optimization version, by calling the decision version repeatedly (a polynomial number of times).

References

[1] R. E. Ladner "On the structure of polynomial time reducibility," J.ACM, 22, pp. 151–171, 1975. Corollary 1.1. ACM site (http://portal.acm.org/citation.cfm?id=321877&dl=ACM&coll=&CFID=15151515&CFTOKEN=6184618).

[2] William I. Gasarch (June 2002). "The P=?NP poll." (http://www.cs.umd.edu/~gasarch/papers/poll.pdf) (PDF). *SIGACT News* **33** (2): 34–47. doi:10.1145/1052796.1052804. . Retrieved 2008-12-29.

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Section 34.2: Polynomial-time verification, pp.979–983.
- Michael Sipser (1997). *Introduction to the Theory of Computation*. PWS Publishing. ISBN 0-534-94728-X. Sections 7.3–7.5 (The Class NP, NP-completeness, Additional NP-complete Problems), pp.241–271.
- David Harel, Yishai Feldman. Algorithmics: The Spirit of Computing, Addison-Wesley, Reading, MA, 3rd edition, 2004.

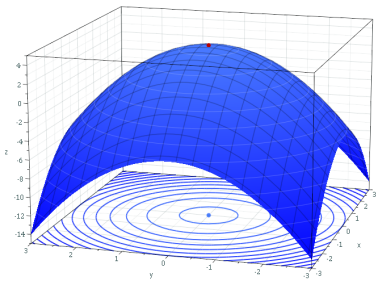
External links

- *Complexity Zoo*: NP (http://qwiki.stanford.edu/wiki/Complexity_Zoo:N#np)
- Graph of NP-complete Problems (http://page.mi.fu-berlin.de/aneumann/npc.html)
- American Scientist primer on traditional and recent complexity theory research: "Accidental Algorithms" (http://www.americanscientist.org/issues/pub/accidental-algorithms/)

Optimization (mathematics)

In mathematics and computer science, **optimization**, or **mathematical programming**, refers to choosing the best element from some set of available alternatives.

In the simplest case, this means solving problems in which one seeks to minimize or maximize a real function by systematically choosing the values of real or integer variables from within an allowed set. This formulation, using a scalar, real-valued objective function, is probably the simplest example; the generalization of optimization theory and techniques to other formulations comprises a large area of applied mathematics. More generally, it means finding "best available" values of some objective function given a defined domain, including a variety of different types of objective functions and different types of domains.



The maximum of a paraboloid

History

The first optimization technique, which is known as steepest descent, goes back to Gauss. Historically, the first term to be introduced was linear programming, which was invented by George Dantzig in the 1940s. The term *programming* in this context does not refer to computer programming (although computers are nowadays used extensively to solve mathematical problems). Instead, the term comes from the use of *program* by the United States military to refer to proposed training and logistics schedules, which were the problems that Dantzig was studying at the time. (Additionally, later on, the use of the term "programming" was apparently important for receiving government funding, as it was associated with high-technology research areas that were considered important.)

Other important mathematicians in the optimization field include:

- Hoang Tuy
- Richard Bellman
- George Dantzig
- Ronald A. Howard
- Leonid Kantorovich
- Narendra Karmarkar
- William Karush
- Leonid Khachiyan
- Bernard Koopman
- Harold Kuhn
- Joseph Louis Lagrange
- László Lovász
- Arkadii Nemirovskii
- Yurii Nesterov
- John von Neumann
- Boris Polyak
- Lev Pontryagin
- James Renegar
- Kees Roos
- Naum Z. Shor
- Michael J. Todd
- Albert Tucker

Major subfields

- Convex programming studies the case when the objective function is convex and the constraints, if any, form a convex set. This can be viewed as a particular case of nonlinear programming or as generalization of linear or convex quadratic programming.
  - Linear programming (LP), a type of convex programming, studies the case in which the objective function *f* is linear and the set of constraints is specified using only linear equalities and inequalities. Such a set is called a polyhedron or a polytope if it is bounded.
  - Second order cone programming (SOCP) is a convex program, and includes certain types of quadratic programs.
  - Semidefinite programming (SDP) is a subfield of convex optimization where the underlying variables are semidefinite matrices. It is generalization of linear and convex quadratic programming.
  - Conic programming is a general form of convex programming. LP, SOCP and SDP can all be viewed as conic programs with the appropriate type of cone.
  - Geometric programming is a technique whereby objective and inequality constraints expressed as posynomials and equality constraints as monomials can be transformed into a convex program.
- Integer programming studies linear programs in which some or all variables are constrained to take on integer values. This is not convex, and in general much more difficult than regular linear programming.
- Quadratic programming allows the objective function to have quadratic terms, while the set A must be specified with linear equalities and inequalities. For specific forms of the quadratic term, this is a type of convex programming.
- Nonlinear programming studies the general case in which the objective function or the constraints or both contain nonlinear parts. This may or may not be a convex program. In general, the convexity of the program affects the difficulty of solving more than the linearity.
- Stochastic programming studies the case in which some of the constraints or parameters depend on random variables.
- Robust programming is, as stochastic programming, an attempt to capture uncertainty in the data underlying the optimization problem. This is not done through the use of random variables, but instead, the problem is solved taking into account inaccuracies in the input data.
- Combinatorial optimization is concerned with problems where the set of feasible solutions is discrete or can be reduced to a discrete one.
- Infinite-dimensional optimization studies the case when the set of feasible solutions is a subset of an infinite-dimensional space, such as a space of functions.
- Heuristic algorithms
  - Metaheuristics
- Constraint satisfaction studies the case in which the objective function *f* is constant (this is used in artificial intelligence, particularly in automated reasoning).
  - Constraint programming.
- Disjunctive programming used where at least one constraint must be satisfied but not all. Of particular use in scheduling.
- Trajectory optimization is the specialty of optimizing trajectories for air and space vehicles.

In a number of subfields, the techniques are designed primarily for optimization in dynamic contexts (that is, decision making over time):

- Calculus of variations seeks to optimize an objective defined over many points in time, by considering how the objective function changes if there is a small change in the choice path.
- Optimal control theory is a generalization of the calculus of variations.

- Dynamic programming studies the case in which the optimization strategy is based on splitting the problem into smaller subproblems. The equation that describes the relationship between these subproblems is called the Bellman equation.
- Mathematical programming with equilibrium constraints is where the constraints include variational inequalities or complementarities.

Multi-objective optimization

Adding more than one objective to an optimization problem adds complexity. For example, if you wanted to optimize a structural design, you would want a design that is both light and rigid. Because these two objectives conflict, a trade-off exists. There will be one lightest design, one stiffest design, and an infinite number of designs that are some compromise of weight and stiffness. This set of trade-off designs is known as a Pareto set. The curve created plotting weight against stiffness of the best designs is known as the Pareto frontier.

A design is judged to be Pareto optimal if it is not dominated by other designs: a Pareto optimal design must be better than another design in at least one aspect. If it is worse than another design in all respects, then it is dominated and is not Pareto optimal.

Multi-modal optimization

Optimization problems are often multi-modal, that is they possess multiple good solutions. They could all be globally good (same cost function value) or there could be a mix of globally good and locally good solutions. Obtaining all (or at least some of) the multiple solutions is the goal of a multi-modal optimizer.

Classical optimization techniques due to their iterative approach do not perform satisfactorily when they are used to obtain multiple solutions, since it is not guaranteed that different solutions will be obtained even with different starting points in multiple runs of the algorithm. Evolutionary Algorithms are however a very popular approach to obtain multiple solutions in a multi-modal optimization task. See Evolutionary multi-modal optimization.

Dimensionless optimization

Dimensionless optimization (DO) is used in design problems, and consists of the following steps:

- Rendering the dimensions of the design dimensionless
- Selecting a local region of the design space to perform analysis on
- Creating an I-optimal design within the local design space
- Forming response surfaces based on the analysis
- Optimizing the design based on the evaluation of the objective function, using the response surface models

Concepts and notation

Optimization problems

An optimization problem can be represented in the following way

> *Given:* a function $f : A \longrightarrow \mathbf{R}$ from some set A to the real numbers
>
> *Sought:* an element x_0 in A such that $f(x_0) \leq f(x)$ for all x in A ("minimization") or such that $f(x_0) \geq f(x)$ for all x in A ("maximization").

Such a formulation is called an **optimization problem** or a **mathematical programming problem** (a term not directly related to computer programming, but still in use for example in linear programming - see History above). Many real-world and theoretical problems may be modeled in this general framework. Problems formulated using this technique in the fields of physics and computer vision may refer to the technique as **energy minimization**, speaking of the value of the function f as representing the energy of the system being modeled.

Typically, *A* is some subset of the Euclidean space $\mathbf{R}^n$, often specified by a set of *constraints*, equalities or inequalities that the members of *A* have to satisfy. The domain *A* of *f* is called the *search space* or the *choice set*, while the elements of *A* are called *candidate solutions* or *feasible solutions*.

The function *f* is called, variously, an **objective function**, **cost function**, **energy function**, or **energy functional**. A feasible solution that minimizes (or maximizes, if that is the goal) the objective function is called an *optimal solution*.

Generally, when the feasible region or the objective function of the problem does not present convexity, there may be several local minima and maxima, where a *local minimum* x^* is defined as a point for which there exists some $\delta > 0$ so that for all x such that

$$\|\mathbf{x} - \mathbf{x}^*\| \leq \delta;$$

the expression

$$f(\mathbf{x}^*) \leq f(\mathbf{x})$$

holds; that is to say, on some region around x^* all of the function values are greater than or equal to the value at that point. Local maxima are defined similarly.

A large number of algorithms proposed for solving non-convex problems – including the majority of commercially available solvers – are not capable of making a distinction between local optimal solutions and rigorous optimal solutions, and will treat the former as actual solutions to the original problem. The branch of applied mathematics and numerical analysis that is concerned with the development of deterministic algorithms that are capable of guaranteeing convergence in finite time to the actual optimal solution of a non-convex problem is called global optimization.

Notation

Optimization problems are often expressed with special notation. Here are some examples.

$$\min_{x \in \mathbb{R}} (x^2 + 1)$$

This asks for the minimum value for the objective function $x^2 + 1$, where x ranges over the real numbers $\mathbb{R}$. The minimum value in this case is 1, occurring at $x = 0$.

$$\max_{x \in \mathbb{R}} 2x$$

This asks for the maximum value for the objective function 2*x*, where *x* ranges over the reals. In this case, there is no such maximum as the objective function is unbounded, so the answer is "infinity" or "undefined".

$$\operatorname{argmin}_{x \in [-\infty, -1]} x^2 + 1$$

This asks for the value (or values) of *x* in the interval $[-\infty, -1]$ that minimizes (or minimize) the objective function $x^2 + 1$ (the actual minimum value of that function does not matter). In this case, the answer is $x = -1$.

$$\operatorname{argmax}_{x \in [-5,5],\, y \in \mathbb{R}} x \cdot \cos(y)$$

This asks for the (x, y) pair (or pairs) that maximizes (or maximize) the value of the objective function $x \cdot \cos(y)$, with the added constraint that x lies in the interval $[-5, 5]$ (again, the actual maximum value of the expression does not matter). In this case, the solutions are the pairs of the form $(5, 2k\pi)$ and $(-5, (2k+1)\pi)$, where k ranges over all integers.

Analytical characterization of optima

Is it possible to satisfy all constraints?

The **satisfiability problem**, also called the **feasibility problem**, is just the problem of finding any feasible solution at all without regard to objective value. This can be regarded as the special case of mathematical optimization where the objective value is the same for every solution, and thus any solution is optimal.

Many optimization algorithms need to start from a feasible point. One way to obtain such a point is to relax the feasibility conditions using a slack variable; with enough slack, any starting point is feasible. Then, minimize that slack variable until slack is null or negative.

Does an optimum exist?

The extreme value theorem of Karl Weierstrass states conditions under which an optimum exists.

How can an optimum be found?

One of Fermat's theorems states that optima of unconstrained problems are found at stationary points, where the first derivative or the gradient of the objective function is zero (see First derivative test). More generally, they may be found at critical points, where the first derivative or gradient of the objective function is zero or is undefined, or on the boundary of the choice set. An equation stating that the first derivative equals zero at an interior optimum is sometimes called a 'first-order condition'.

Optima of inequality-constrained problems are instead found by the Lagrange multiplier method. This method calculates a system of inequalities called the 'Karush-Kuhn-Tucker conditions' or 'complementary slackness conditions', which may then be used to calculate the optimum.

While the first derivative test identifies points that might be optima, it cannot distinguish a point which is a minimum from one that is a maximum or one that is neither. When the objective function is twice differentiable, these cases can be distinguished by checking the second derivative or the matrix of second derivatives (called the Hessian matrix) in unconstrained problems, or a matrix of second derivatives of the objective function and the constraints called the bordered Hessian. The conditions that distinguish maxima and minima from other stationary points are sometimes called 'second-order conditions' (see 'Second derivative test').

How does the optimum change if the problem changes?

The envelope theorem describes how the value of an optimal solution changes when an underlying parameter changes.

The maximum theorem of Claude Berge (1963) describes the continuity of the optimal solution as a function of underlying parameters.

Computational optimization techniques

Crudely all the methods are divided according to variables called:-
SVO:- Single-variable optimization
MVO:- Multi-variable optimization
For twice-differentiable functions, unconstrained problems can be solved by finding the points where the gradient of the objective function is zero (that is, the stationary points) and using the Hessian matrix to classify the type of each point. If the Hessian is positive definite, the point is a local minimum, if negative definite, a local maximum, and if indefinite it is some kind of saddle point.

The existence of derivatives is not always assumed and many methods were devised for specific situations. The basic classes of methods, based on smoothness of the objective function, are:

- Combinatorial methods
- Derivative-free methods
- First-order methods
- Second-order methods

Actual methods falling somewhere among the categories above include:

- Bundle methods
- Conjugate gradient method
- Ellipsoid method
- Frank–Wolfe method
- Gradient descent aka steepest descent or steepest ascent
- Interior point methods
- Line search - a technique for one dimensional optimization, usually used as a subroutine for other, more general techniques.
- Nelder-Mead method aka the Amoeba method
- Newton's method
- Quasi-Newton methods
- Simplex method
- Subgradient method - similar to gradient method in case there are no gradients

Should the objective function be convex over the region of interest, then any local minimum will also be a global minimum. There exist robust, fast numerical techniques for optimizing twice differentiable convex functions.

Constrained problems can often be transformed into unconstrained problems with the help of Lagrange multipliers.

Here are a few other popular methods:

- Filled function method
- Ant colony optimization
- Beam search
- Bees algorithm
- Differential evolution
- Dynamic relaxation
- Evolution strategy
- Genetic algorithms
- Harmony search
- Hill climbing
- Particle swarm optimization
- Quantum annealing
- Simulated annealing
- Stochastic tunneling
- Tabu search
- IOSO

Applications

Problems in rigid body dynamics (in particular articulated rigid body dynamics) often require mathematical programming techniques, since you can view rigid body dynamics as attempting to solve an ordinary differential equation on a constraint manifold; the constraints are various nonlinear geometric constraints such as "these two points must always coincide", "this surface must not penetrate any other", or "this point must always lie somewhere on this curve". Also, the problem of computing contact forces can be done by solving a linear complementarity problem, which can also be viewed as a QP (quadratic programming) problem.

Many design problems can also be expressed as optimization programs. This application is called design optimization. One recent and growing subset of this field is multidisciplinary design optimization, which, while useful in many problems, has in particular been applied to aerospace engineering problems.

Economics also relies heavily on mathematical programming. An often studied problem in microeconomics, the utility maximization problem, and its dual problem the Expenditure minimization problem, are economic optimization problems. Consumers and firms are assumed to maximize their utility/profit. Also, agents are most frequently assumed to be risk-averse thereby wishing to minimize whatever risk they might be exposed to. Asset prices are also explained using optimization though the underlying theory is more complicated than simple utility or profit optimization. Trade theory also uses optimization to explain trade patterns between nations.

Another field that uses optimization techniques extensively is operations research.

See also

- Curve fitting
- Arg max
- Brachistochrone
- Dynamic programming
- Fuzzy logic
- Game theory
- Goal programming
- Important publications in optimization
- Interior point methods
- Least squares
- Numerical analysis
- Operations research
- Optimization problem
- Optimization algorithms
- Mathematical optimization software
- Process optimization
- Radial basis function
- Random optimization
- Simplex algorithm
- Topkis's theorem
- Variational calculus
- Variational inequality
- Interval finite element

Solvers

- Comet
- CPLEX
- FortSP - solver for stochastic programming problems
- Gurobi
- IMSL Numerical Libraries are collections of math and statistical algorithms available in C/C++, Fortran, Java and C#/.NET. Optimization routines in the IMSL Libraries include unconstrained, linearly and nonlinearly constrained minimizations, and linear programming algorithms.
- IPOPT - an open-source primal-dual interior point method NLP solver which handles sparse matrices
- KNITRO - solver for nonlinear optimization problems
- Mathematica - handles linear programming, integer programming and constrained non-linear optimization problems
- Merlin - A Fortran-77, user friendly open source software package, for non-linear optimization with bound constraints. URL: http://merlin.cs.uoi.gr
- MINUIT
- OpenOpt - a free optimization framework written in Python and NumPy, connects to tens of solvers, can involve Automatic differentiation
- Opt++ - An object-oriented package from Lawrence Berkeley and Sandia National Labs, used for nonlinear optimization. URL: http://acts.nersc.gov/opt++/
- SNOPT

References

- Mordecai Avriel (2003). *Nonlinear Programming: Analysis and Methods.* Dover Publishing. ISBN 0-486-43227-0.
- Stephen Boyd and Lieven Vandenberghe (2004). Convex Optimization [1], Cambridge University Press. ISBN 0-521-83378-7.
- Elster K.-H. (1993), Modern Mathematical Methods of Optimization, Vch Pub. ISBN 3-05-501452-9.
- Jorge Nocedal and Stephen J. Wright (2006). Numerical Optimization [2], Springer. ISBN 0-387-30303-0.
- Panos Y. Papalambros and Douglass J. Wilde (2000). Principles of Optimal Design : Modeling and Computation [3], Cambridge University Press. ISBN 0-521-62727-3.
- Rowe W.B.; O'Donoghue J.P. Cameron, A; (1970) Optimization of externally pressurized bearing for minimum power and low temperature rise. Tribology (London)
- Yang X.-S. (2008), Introduction to Mathematical Optimization: From Linear Programming to Metaheuristics, Cambridge Int. Science Publishing. ISBN 1-904602-82-7.

External links

- COIN-OR [4] — Computational Infrastructure for Operations Research
- Decision Tree for Optimization Software [5] Links to optimization source codes
- Global optimization [6]
- Mathematical Programming Glossary [2]
- Mathematical Programming Society [7]
- NEOS Guide [8] currently being replaced by the NEOS Wiki [9]
- Optimization Online [10] A repository for optimization e-prints
- Optimization Related Links [11]
- Convex Optimization I [12] EE364a: Course from Stanford University
- Convex Optimization – Boyd and Vandenberghe [13] Book on Convex Optimization
- Simplemax Online Optimization Services [14] Web applications to access nonlinear optimization services

Solvers:

- APOPT [15] - large-scale nonlinear programming
- Free Optimization Software by Systems Optimization Laboratory, Stanford University [16]
- Moocho [17] - a very flexible open-source NLP solver
- TANGO Project [18] - Trustable Algorithms for Nonlinear General Optimization - Fortran

Libraries:

- IMSL Numerical Libraries Collections of math and statistical algorithms available in C/C++, Fortran, Java and C#/.NET. Optimization routines in the IMSL Libraries include unconstrained, linearly and nonlinearly constrained minimizations, and linear programming algorithms.
- ALGLIB [19] Open source optimization routines (unconstrained and bound constrained optimization). C++, C#, Delphi, Visual Basic.
- IOptLib (Investigative Optimization Library) [20] - a free open source library for development of optimization algorithms (ANSI C).
- OAT (Optimization Algorithm Toolkit) [21] - a set of standard optimization algorithms and problems in Java.
- OOL (Open Optimization library) [22] - a set of optimization routines in C.

References

[1] http://www.stanford.edu/~boyd/cvxbook/
[2] http://www.ece.northwestern.edu/~nocedal/book/num-opt.html
[3] http://www.optimaldesign.org/
[4] http://www.coin-or.org/
[5] http://plato.asu.edu/guide.html
[6] http://www.mat.univie.ac.at/%7Eneum/glopt.html
[7] http://www.mathprog.org/
[8] http://www-fp.mcs.anl.gov/otc/Guide/index.html
[9] http://wiki.mcs.anl.gov/neos
[10] http://www.optimization-online.org
[11] http://www2.arnes.si/%7Eljc3m2/igor/links.html
[12] http://see.stanford.edu/see/courseinfo.aspx?coll=2db7ced4-39d1-4fdb-90e8-364129597c87
[13] http://www.stanford.edu/~boyd/cvxbook
[14] http://simplemax.net
[15] http://wiki.mcs.anl.gov/NEOS/index.php/APOPT
[16] http://www.stanford.edu/group/SOL/software.html
[17] http://trilinos.sandia.gov/packages/moocho/
[18] http://www.ime.usp.br/~egbirgin/tango/
[19] http://www.alglib.net/optimization/
[20] http://www2.arnes.si/~ljc3m2/igor/ioptlib/
[21] http://optalgtoolkit.sourceforge.net/
[22] http://ool.sourceforge.net/

P (complexity)

In computational complexity theory, **P**, also known as **PTIME** or **DTIME**($n^{O(1)}$), is one of the most fundamental complexity classes. It contains all decision problems which can be solved by a deterministic Turing machine using a polynomial amount of computation time, or polynomial time.

Cobham's thesis holds that **P** is the class of computational problems which are "efficiently solvable" or "tractable"; in practice, some problems not known to be in **P** have practical solutions, and some that are in **P** do not, but this is a useful rule of thumb.

Definition

A language *L* is in **P** if and only if there exists a deterministic Turing machine *M*, such that

- *M* runs for polynomial time on all inputs
- For all *x* in *L*, *M* outputs 1
- For all *x* not in *L*, *M* outputs 0

P can also be viewed as a uniform family of boolean circuits. A language *L* is in **P** if and only if there exists a polynomial-time uniform family of boolean circuits $\{C_n : n \in \mathbb{N}\}$, such that

- For all $n \in \mathbb{N}$, C_n takes *n* bits as input and outputs 1 bit
- For all *x* in *L*, $C_{|x|}(x) = 1$
- For all *x* not in *L*, $C_{|x|}(x) = 0$

The circuit definition can be weakened to use only a logspace uniform family without changing the complexity class **P**.

Notable problems in P

P is known to contain many natural problems, including the decision versions of linear programming, calculating the greatest common divisor, and finding a maximum matching. In 2002, it was shown that the problem of determining if a number is prime is in **P**.[1] The related class of function problems is **FP**.

Several natural problems are complete for **P**, including *st*-connectivity (or reachability) on alternating graphs.[2] The article on **P**-complete problems lists further relevant problems in **P**.

Relationships to other classes

A generalization of **P** is **NP**, which is the class of languages decidable in polynomial time on a non-deterministic Turing machine. We then trivially have **P** is a subset of **NP**. Though unproven, most experts believe this is a strict subset.[3]

P is also known to be at least as large as **L**, the class of problems decidable in a logarithmic amount of memory space. A decider using $O(\log n)$space cannot use more than $2^{O(\log n)} = n^{O(1)}$time, because this is the total number of possible configurations; thus, **L** is a subset of **P**. Another important problem is whether **L = P**. We do know that **P = AL**, the set of problems solvable in logarithmic memory by alternating Turing machines. **P** is also known to be no larger than **PSPACE**, the class of problems decidable in polynomial space. Again, whether **P = PSPACE** is an open problem. To summarize:

$$\mathrm{L} \subseteq \mathrm{AL} = \mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{PSPACE} \subseteq \mathrm{EXPTIME}.$$

Here, **EXPTIME** is the class of problems solvable in exponential time. Of all the classes shown above, only two strict containments are known:

- **P** is strictly contained in **EXPTIME**. Consequently, all **EXPTIME**-hard problems lie outside **P**, and at least one of the containments to the right of **P** above is strict (in fact, it is widely believed that all three are strict).
- **L** is strictly contained in **PSPACE**.

The most difficult problems in P are P-complete problems.

Another generalization of **P** is **P/poly**, or **Nonuniform Polynomial-Time**. If a problem is in **P/poly**, then it can be solved in deterministic polynomial time provided that an advice string is given that depends only on the length of the input. Unlike for **NP**, however, the polynomial-time machine doesn't need to detect fraudulent advice strings; it is not a verifier. **P/poly** is a large class containing nearly all practical algorithms, including all of **BPP**. If it contains **NP**, then the polynomial hierarchy collapses to the second level. On the other hand, it also contains some impractical algorithms, including some undecidable problems such as the unary version of any undecidable problem.

In 1999, Jin-Yi Cai and D. Sivakumar, building on work by Mitsunori Ogihara, showed that if there exists a sparse language which is **P-complete**, then **L = P**.[4]

Properties

Polynomial-time algorithms are closed under composition. Intuitively, this says that if one writes a function which is polynomial-time assuming that function calls are constant-time, and if those called functions themselves require polynomial time, then the entire algorithm takes polynomial time. One consequence of this is that **P** is low for itself. This is also one of the main reasons that **P** is considered to be a machine-independent class; any machine "feature", such as random access, which can be simulated in polynomial time can simply be composed with the main polynomial-time algorithm to reduce it to a polynomial-time algorithm on a more basic machine.

Pure existence proofs of polynomial-time algorithms

Some problems are known to be solvable in polynomial-time, but no concrete algorithm is known for solving them. For example, the Robertson-Seymour theorem guarantees that there is a finite list of forbidden minors that characterizes (for example) the set of graphs that can be embedded on a torus; moreover, Robertson and Seymour showed that there is an $O(n^3)$ algorithm for determining whether a graph has a given graph as a minor. This yields a nonconstructive proof that there is a polynomial-time algorithm for determining if a given graph can be embedded on a torus, despite the fact that no concrete algorithm is known for this problem.

Alternative characterizations

In descriptive complexity, **P** can be described as the problems expressible in FO (LFP), the class of first-order logic with a least fixed point operator added to it. In Immerman's 1999 textbook on descriptive complexity,[5] Immerman ascribes this result to Vardi[6] and to Immerman.[7]

History

Kozen[8] states that Cobham and Edmonds are "generally credited with the invention of the notion of polynomial time." Cobham invented the class as a robust way of characterizing efficient algorithms, leading to Cobham's thesis. However, H. C. Pocklington, in a 1910 paper,[9] [10] analyzed two algorithms for solving quadratic congruences, and observed that one took time "proportional to a power of the logarithm of the modulus" and contrasted this with one that took time proportional "to the modulus itself or its square root", thus explicitly drawing a distinction between an algorithm that ran in polynomial time versus one that did not.

References

- Cobham, Alan (1965). "The intrinsic computational difficulty of functions". *Proc. Logic, Methodology, and Philosophy of Science II*. North Holland.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Section 34.1: Polynomial time, pp.971–979.
- Papadimitriou, Christos H. (1994). *Computational complexity*. Reading, Mass.: Addison-Wesley. ISBN 0-201-53082-1.
- Sipser, Michael (2006). *Introduction to the Theory of Computation*. Course Technology Inc. ISBN 0-619-21764-2. Section 7.2: The Class P, pp.234–241.

External links

- *Complexity Zoo*: Class P [11]
- *Complexity Zoo*: Class P/poly [12]

References

[1] Manindra Agrawal, Neeraj Kayal, Nitin Saxena, " PRIMES is in P (http://www.cse.iitk.ac.in/users/manindra/algebra/primality_v6.pdf)", *Annals of Mathematics* 160 (2004), no. 2, pp. 781–793.

[2] Immerman, Neil (1999). *Descriptive Complexity*. New York: Springer-Verlag. ISBN 0-387-98600-6.

[3] Johnsonbaugh, Richard; Schaefer, Marcus, *Algorithms*, 2004 Pearson Education, page 458, ISBN 0-02-360692-4

[4] Jin-Yi Cai and D. Sivakumar. Sparse hard sets for P: resolution of a conjecture of Hartmanis. *Journal of Computer and System Sciences*, volume 58, issue 2, pp.280–296. 1999. ISSN:0022-0000. At Citeseer (http://citeseer.ist.psu.edu/501645.html)

[5] Immerman, Neil (1999). *Descriptive Complexity*. New York: Springer-Verlag. p. 66. ISBN 0-387-98600-6.

[6] Vardi, Moshe Y. (1982). "The Complexity of Relational Query Languages". *STOC '82: Proceedings of the fourteenth annual ACM symposium on Theory of computing*. pp. 137–146. doi:10.1145/800070.802186.

[7] Immerman, Neil (1982). "Relational Queries Computable in Polynomial Time". *STOC '82: Proceedings of the fourteenth annual ACM symposium on Theory of computing*. pp. 147–152. doi:10.1145/800070.802187. Revised version in *Information and Control*, 68 (1986), 86-104.

[8] Kozen, Dexter C. (2006). *Theory of Computation*. Springer. p. 4. ISBN 1-84628-297-7.

[9] Pocklington, H. C. (1910-2). "The determination of the exponent to which a number belongs, the practical solution of certain congruences, and the law of quadratic reciprocity". *Proc. Cambridge Phil. Soc.* **16**: pp. 1–5.

[10] Gautschi, Walter (1994). *Mathematics of computation, 1943-1993: a half-century of computational mathematics: Mathematics of Computation 50th Anniversary Symposium, August 9-13, 1993, Vancouver, British Columbia*. Providence, RI: American Mathematical Society. pp. 503–504. ISBN 0-8218-0291-7.

[11] http://qwiki.stanford.edu/wiki/Complexity_Zoo:P#p

[12] http://qwiki.stanford.edu/wiki/Complexity_Zoo:P#ppoly

P versus NP problem

Millennium Prize Problems
P versus NP problem
Hodge conjecture
Poincaré conjecture (solution)
Riemann hypothesis
Yang–Mills existence and mass gap
Navier–Stokes existence and smoothness
Birch and Swinnerton-Dyer conjecture

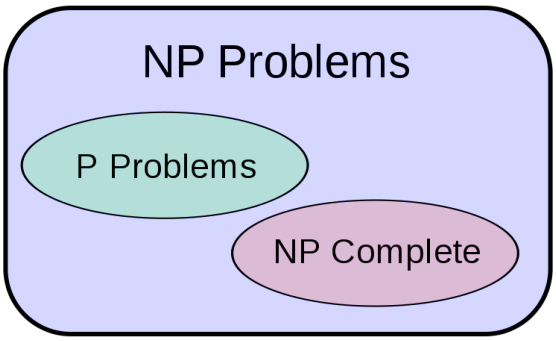


Diagram of complexity classes provided that **P** ≠ **NP**. The existence of problems outside both **P** and **NP**-complete in this case was established by Ladner's theorem.[1]

The relationship between the complexity classes **P** and **NP** is an unsolved problem in theoretical computer science, and is considered by many theoretical computer scientists to be the most important problem in the field.[2] The Clay Mathematics Institute, which is dedicated to increasing and disseminating mathematical knowledge, has included it in its list of Millennium Prize Problems; anyone that provides a satisfactory solution to the problem may be entitled to a million dollar prize.[3] [4]

In essence, the question **P** = **NP**? asks: if 'yes'-answers to a 'yes'-or-'no'-question can be *verified* "quickly" can the answers themselves also be *computed* "quickly"? The theoretical notion of "quick" used here is that of an algorithm that runs in polynomial time, which sometimes but not always corresponds to an algorithm that is fast in practice.

Consider the subset sum problem, an example of a problem which is easy to verify but whose answer is suspected to be theoretically difficult to compute. Given a set of integers, does some nonempty subset of them sum to 0? For instance, does a subset of the set {−2, −3, 15, 14, 7, −10} add up to 0? The answer "yes, because {−2, −3, −10, 15} add up to zero", can be quickly verified with three additions. However, finding such a subset in the first place could take more time. The information needed to verify a positive answer is also called a *certificate*. Given the right certificates, "yes" answers to our problem can be verified in polynomial time, so this problem is in **NP**.

An answer to the **P** = **NP** question would determine whether problems like the subset-sum problem that can be verified in polynomial time can also be solved in polynomial time. If it turned out that **P** does not equal **NP**, it would mean that some **NP** problems are harder to compute than to verify in that they could not be solved in polynomial time but the answer can be verified in polynomial time.

The restriction to yes/no problems is unimportant; when more complicated answers are allowed the problem becomes **FP** = **FNP**, and that is proven to be equivalent to **P** = **NP**.[5]

Context of the problem

The relation between the complexity classes **P** and **NP** is studied in computational complexity theory, the part of the theory of computation dealing with the resources required during computation to solve a given problem. The most common resources are time (how many steps it takes to solve a problem) and space (how much memory it takes to solve a problem).

In such analysis, a model of the computer for which time must be analyzed is required. Typically such models assume that the computer is *deterministic* (given the computer's present state and any inputs, there is only one possible action that the computer might take) and *sequential* (it performs actions one after the other).

In this theory, the class **P** consists of all those *decision problems* (defined below) that can be solved on a deterministic sequential machine in an amount of time that is polynomial in the size of the input; the class **NP** consists of all those decision problems whose positive solutions can be verified in polynomial time given the right information, or equivalently, whose solution can be found in polynomial time on a non-deterministic machine.[6] Arguably the biggest open question in theoretical computer science concerns the relationship between those two classes:

Is **P** equal to **NP**?

In a 2002 poll of 100 researchers 61 believed the answer to be no, 9 believed the answer is yes, and 22 were unsure; 8 believed the question may be independent of the currently accepted axioms and so impossible to prove or disprove.[7]

Example

Let $COMPOSITE = \{x \in N : x = pq \text{ for integers } p, q > 1\}$ and $R = \{(x, y) \in N \times N : 1 < y \leq \sqrt{x}\ ;\ y \text{ divides } x\}$.
Clearly, the question of whether a given x is a composite is equivalent to the question of whether x is a member of $COMPOSITE$. It can be shown that $COMPOSITE \in \mathbf{NP}$ by verifying that $COMPOSITE$ satisfies the above definition.

$COMPOSITE$ also happens to be in **P**.[8] [9]

NP-complete

To attack the **P** = **NP** question the concept of **NP**-completeness is very useful. Informally the **NP**-complete problems are the "toughest" problems in **NP** in the sense that they are the ones most likely not to be in **P**. **NP**-complete problems are those **NP**-hard problems which are in **NP**, where **NP**-hard problems are those to which *any* problem in **NP** can be reduced in polynomial time. For instance, the decision problem version of the travelling salesman problem is **NP**-complete, so *any* instance of *any* problem in **NP** can be transformed mechanically into an instance of the traveling salesman problem, in polynomial time. The traveling salesman problem is one of many such **NP**-complete problems. If any **NP**-complete problem is in **P**, then it would follow that **P** = **NP**. Unfortunately, many important problems have been shown to be **NP**-complete and as of 2010, not a single fast algorithm for any of them is known.

Based on the definition alone it's not obvious that **NP**-complete problems exist. A trivial and contrived **NP**-complete problem can be formulated as: given a description of a Turing machine M guaranteed to halt in polynomial time, does there exist a polynomial-size input that M will accept?[10] It is in **NP** because (given an input) it is simple to check whether or not M accepts the input by simulating M; it is **NP**-hard because the verifier for any particular instance of a problem in **NP** can be encoded as a polynomial-time machine M that takes the solution to be verified as input. Then the question of whether the instance is a yes or no instance is determined by whether a valid input exists.

The first natural problem proven to be **NP**-complete was the Boolean satisfiability problem. This result came to be known as Cook–Levin theorem; its proof that satisfiability is NP-complete contains technical details about Turing machines as they relate to the definition of **NP**. However, after this problem was proved to be NP-complete, proof by reduction provided a simpler way to show that many other problems are in this class. Thus, a vast class of seemingly unrelated problems are all reducible to one another, and are in a sense the "same problem".

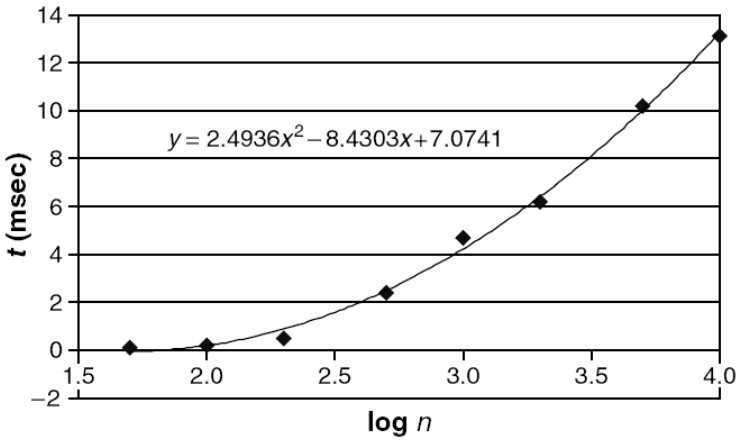
Still harder problems

Although it is unknown whether **P** = **NP**, problems outside of **P** are known. A number of succinct problems (problems which operate not on normal input but on a computational description of the input) are known to be **EXPTIME**-complete. Because it can be shown that **P** $\subsetneq$ **EXPTIME**, these problems are outside **P**, and so require more than polynomial time. In fact, by the time hierarchy theorem, they cannot be solved in significantly less than exponential time. Examples include finding a perfect strategy for chess (on an N×N board)[11] and some other board games.[12]

The problem of deciding the truth of a statement in Presburger arithmetic requires even more time. Fischer and Rabin proved in 1974 that every algorithm which decides the truth of Presburger statements has a runtime of at least $2^{2^{cn}}$ for some constant *c*. Here, *n* is the length of the Presburger statement. Hence, the problem is known to need more than exponential run time. Even more difficult are the undecidable problems, such as the halting problem. They cannot be completely solved by any algorithm, in the sense that for any particular algorithm there is at least one input for which that algorithm will not produce the right answer; it will either produce the wrong answer, finish without giving a conclusive answer, or otherwise run forever without producing any answer at all.

Does P mean "easy"?

All of the above discussion has assumed that **P** means "easy" and "not in **P**" means "hard". This assumption, known as *Cobham's thesis*, though a common and reasonably accurate assumption in complexity theory, is not always true in practice; the size of constant factors or exponents may have practical importance, or there may be solutions that work for situations encountered in practice despite having poor worst-case performance in theory (this is the case for instance for the simplex algorithm in linear programming). Other solutions violate the Turing machine model on which **P** and **NP** are defined by introducing concepts like randomness and quantum computation.



The graph shows time (average of 100 instances in msec using a 933 MHz Pentium III) vs.problem size for knapsack problems for a state-of-the-art specialized algorithm. Quadratic fit suggests that empirical algorithmic complexity for instances with 50–10,000 variables is $O((\log n)^2)$.[13]

Because of these factors even if a problem is shown to be NP-complete, and even if **P** ≠ **NP**, there may still be effective approaches to tackling the problem in practice. There are algorithms for many NP-complete problems, such as the knapsack problem, the travelling salesman problem and the boolean satisfiability problem, that can solve to optimality many real-world instances in reasonable time. The empirical average complexity (time vs. problem size) of such algorithms can be surprisingly low.

Why many computer scientists think P ≠ NP

According to a poll[7] many computer scientists believe that **P ≠ NP**. A key reason for this belief is that after decades of studying these problems no one has been able to find a polynomial-time algorithm for any of more than 3000 important known **NP**-complete problems (see List of NP-complete problems). These algorithms were sought long before the concept of **NP**-completeness was even defined (Karp's 21 NP-complete problems, among the first found, were all well-known existing problems at the time they were shown to be NP-complete). Furthermore, the result **P = NP** would imply many other startling results that are currently believed to be false, such as **NP = co-NP** and **P = PH**.

It is also intuitively argued that the existence of problems that are hard to solve but for which the solutions are easy to verify matches real-world experience.[14]

> If P = NP, then the world would be a profoundly different place than we usually assume it to be. There would be no special value in “creative leaps,” no fundamental gap between solving a problem and recognizing the solution once it’s found. Everyone who could appreciate a symphony would be Mozart; everyone who could follow a step-by-step argument would be Gauss...
>
> — Scott Aaronson, MIT

On the other hand, some researchers believe that we are overconfident in **P ≠ NP** and should explore proofs of **P = NP** as well. For example, in 2002 these statements were made:[7]

> The main argument in favor of **P ≠ NP** is the total lack of fundamental progress in the area of exhaustive search. This is, in my opinion, a very weak argument. The space of algorithms is very large and we are only at the beginning of its exploration. [. . .] The resolution of Fermat's Last Theorem also shows that very simple questions may be settled only by very deep theories.
>
> —Moshe Y. Vardi, Rice University
>
> Being attached to a speculation is not a good guide to research planning. One should always try both directions of every problem. Prejudice has caused famous mathematicians to fail to solve famous problems whose solution was opposite to their expectations, even though they had developed all the methods required.
>
> —Anil Nerode, Cornell University

Consequences of proof

One of the reasons the problem attracts so much attention is the consequences of the answer.

A proof of **P = NP** could have stunning practical consequences, if the proof leads to efficient methods for solving some of the important problems in NP. It is also possible that a proof would not lead directly to efficient methods, perhaps if the proof is non-constructive, or the size of the bounding polynomial is too big to be efficient in practice. Various NP-complete problems are fundamental in many fields. There are enormous positive consequences that would follow from rendering tractable many currently mathematically intractable problems. For instance, many problems in operations research are NP-complete, such as some types of integer programming, and the travelling salesman problem, to name two of the most famous examples. Efficient solutions to these problems would have enormous implications for logistics. Many other important problems, such as some problems in protein structure prediction are also **NP**-complete;[15] if these problems were efficiently solvable it could spur considerable advances in biology.

But such changes may pale in significance compared to the revolution an efficient method for solving NP-complete problems would cause in mathematics itself. According to Stephen Cook,[4]

> ...it would transform mathematics by allowing a computer to find a formal proof of any theorem which has a proof of a reasonable length, since formal proofs can easily be recognized in polynomial time. Example problems may well include all of the CMI prize problems.

Research mathematicians spend their careers trying to prove theorems, and some proofs have taken decades or even centuries to find after problems have been stated – for instance, Fermat's Last Theorem took over three centuries to prove. A method that is guaranteed to find proofs to theorems, should one exist of a "reasonable" size, would essentially end this struggle.

A proof that showed that **P ≠ NP**, while lacking the practical computational benefits of a proof that **P = NP**, would also represent a very significant advance in computational complexity theory and provide guidance for future research. It would allow one to show in a formal way that many common problems cannot be solved efficiently, so that the attention of researchers can be focused on partial solutions or solutions to other problems. Due to widespread belief in **P ≠ NP**, much of this focusing of research has already taken place.[16]

Results about difficulty of proof

The Clay Mathematics Institute million-dollar prize and a huge amount of dedicated research with no substantial results suggest that the problem is difficult. Some of the most fruitful research related to the **P = NP** problem has been in showing that existing proof techniques are not powerful enough to answer the question, thus suggesting that novel technical approaches are required.

As additional evidence for the difficulty of the problem, essentially all known proof techniques in computational complexity theory fall into one of the following classifications, each of which is known to be insufficient to prove that **P ≠ NP**:

- **Relativizing proofs:** Imagine a world where every algorithm is allowed to make queries to some fixed subroutine called an oracle, and the running time of the oracle is not counted against the running time of the algorithm. Most proofs (especially classical ones) apply uniformly in a world with oracles regardless of what the oracle does. These proofs are called *relativizing*. In 1975, Baker, Gill, and Solovay showed that **P = NP** with respect to some oracles, while **P ≠ NP** for other oracles.[17] Since relativizing proofs can only prove statements that are uniformly true with respect to all possible oracles, this showed that relativizing techniques cannot resolve **P = NP**.
- **Natural proofs:** In 1993, Alexander Razborov and Steven Rudich defined a general class of proof techniques for circuit complexity lower bounds, called *natural proofs*. At the time all previously known circuit lower bounds were natural, and circuit complexity was considered a very promising approach for resolving **P = NP**. However, Razborov and Rudich showed that in order to prove **P ≠ NP** using a natural proof, one necessarily must also prove an even stronger statement, which is believed to be false. Thus it is unlikely that natural proofs alone can resolve **P = NP**.
- **Algebrizing proofs:** After the Baker-Gill-Solovay result, new non-relativizing proof techniques were successfully used to prove that IP = PSPACE. However, in 2008, Scott Aaronson and Avi Wigderson showed that the main technical tool used in the **IP = PSPACE** proof, known as *arithmetization*, was also insufficient to resolve **P = NP**.[18]

These barriers are another reason why **NP**-complete problems are useful: if a polynomial-time algorithm can be demonstrated for an **NP**-complete problem, this would solve the **P = NP** problem in a way which is not excluded by the above results.

These barriers have also led some computer scientists to suggest that the P versus NP problem may be independent of standard axiom systems like ZFC (cannot be proved or disproved within them). The interpretation of an independence result could be that either no polynomial-time algorithm exist for any NP-complete problem, but such a proof cannot be constructed in (say) ZFC, or that polynomial-time algorithms for NP-complete problems may exist, but it's impossible to prove (in ZFC) that such algorithms are correct.[19] However, if the problem cannot be decided even with much weaker assumptions extending the Peano axioms (PA) for integer arithmetic, then there would necessarily exist nearly-polynomial-time algorithms for every problem in NP.[20] Therefore, if one believes (as most complexity theorists do) that problems in NP do not have efficient algorithms, it would follow that such notions of independence cannot be possible. Additionally, this result implies that proving independence from PA or

ZFC using currently known techniques is no easier than proving the existence of efficient algorithms for all problems in NP.

Logical characterizations

The **P = NP** problem can be restated in terms of the expressibility of certain classes of logical statements, as a result of work in descriptive complexity. All languages (of finite structures with a fixed signature including a linear order relation) in **P** can be expressed in first-order logic with the addition of a suitable least fixed point operator (effectively, this, in combination with the order, allows the definition of recursive functions); indeed, (as long as the signature contains at least one predicate or function in addition to the distinguished order relation [so that the amount of space taken to store such finite structures is actually polynomial in the number of elements in the structure]), this precisely characterizes **P**. Similarly, **NP** is the set of languages expressible in existential second-order logic — that is, second-order logic restricted to exclude universal quantification over relations, functions, and subsets. The languages in the polynomial hierarchy, **PH**, correspond to all of second-order logic. Thus, the question "is **P** a proper subset of **NP**" can be reformulated as "is existential second-order logic able to describe languages (of finite linearly ordered structures with nontrivial signature) that first-order logic with least fixed point cannot?". The word "existential" can even be dropped from the previous characterization, since **P = NP** if and only if **P = PH** (as the former would establish that **NP = co-NP**, which in turn would imply that **NP = PH**). PSPACE = NPSPACE as established Savitch's theorem, this follows directly from the fact that the square of a polynomial function is still a polynomial function. However, it is believed, but not proven, a similar relationship may not exist between the polynomial time complexity classes, P and NP so the question is still open.

Polynomial-time algorithms

No algorithm for any **NP**-complete problem is known to run in polynomial time. However, there are algorithms for **NP**-complete problems with the property that if **P = NP**, then the algorithm runs in polynomial time (although with enormous constants, making the algorithm impractical). The following algorithm, due to Levin, is such an example. It correctly accepts the **NP**-complete language SUBSET-SUM, and runs in polynomial time if and only if **P = NP**:

```
// Algorithm that accepts the NP-complete language SUBSET-SUM.
//
// This is a polynomial-time algorithm if and only if P=NP.
//
// "Polynomial-time" means it returns "yes" in polynomial time when
// the answer should be "yes", and runs forever when it is "no".
//
// Input:  S = a finite set of integers
// Output: "yes" if any subset of S adds up to 0.
//         Runs forever with no output otherwise.
// Note:  "Program number P" is the program obtained by
//         writing the integer P in binary, then
//         considering that string of bits to be a
//         program. Every possible program can be
//         generated this way, though most do nothing
//         because of syntax errors.

FOR N = 1...infinity
    FOR P = 1...N
        Run program number P for N steps with input S
```

```
            IF the program outputs a list of distinct integers
                AND the integers are all in S
                AND the integers sum to 0

            THEN
                OUTPUT "yes" and HALT
```

If, and only if, **P = NP**, then this is a polynomial-time algorithm accepting an **NP**-complete language. "Accepting" means it gives "yes" answers in polynomial time, but is allowed to run forever when the answer is "no".

Note that this is enormously impractical, even if **P = NP**. If the shortest program that can solve SUBSET-SUM in polynomial time is *b* bits long, the above algorithm will try $2^b - 1$ other programs first.

Perhaps we want to "solve" the SUBSET-SUM problem, rather than just "accept" the SUBSET-SUM language. That means we want the algorithm to always halt and return a "yes" or "no" answer. If **P = NP**, then there is an algorithm which does this in polynomial time, which uses some polynomial-time verification method that there is no subset sum in the algorithm above. Another algorithm that is obtained by replacing the IF statement in the above algorithm with this:

```
            IF the program outputs a complete math proof
                AND each step of the proof is legal
                AND the conclusion is that S does (or does not) have a
subset summing to 0
            THEN
                OUTPUT "yes" (or "no") and HALT
```

Formal definitions for P and NP

Conceptually a *decision problem* is a problem that takes as input some string, and outputs "yes" or "no". If there is an algorithm (say a Turing machine, or a computer program with unbounded memory) which is able to produce the correct answer for any input string of length n in at most $c \cdot n^k$ steps, where k and c are constants independent of the input string, then we say that the problem can be solved in *polynomial time* and we place it in the class **P**. Formally, **P** is defined as the set of all languages which can be decided by a deterministic polynomial-time Turing machine. That is,

P = $\{L : L = L(M) \text{ for some deterministic polynomial-time Turing machine } M\}$

where $L(M) = \{w \in \Sigma^* : M \text{ accepts } w\}$

and a deterministic polynomial-time Turing machine is a deterministic Turing machine M which satisfies the following two conditions:

1. M halts on all input w; and
2. there exists $k \in N$ such that $T_M(n) \in O\left(n^k\right)$,

   where $T_M(n) = \max\{t_M(w) : w \in \Sigma^*, |w| = n\}$

   and $t_M(w) =$ number of steps M takes to halt on input w.

NP can be defined similarly using nondeterministic Turing machines (the traditional way). However, a modern approach to define **NP** is to use the concept of *certificate* and *verifier*. Formally, **NP** is defined as the set of languages over a finite alphabet that have a verifier that runs in polynomial time, where the notion of "verifier" is defined as follows.

Let L be a language over a finite alphabet, Σ.

$L \in \mathbf{NP}$ if, and only if, there exists a binary relation $R \subset \Sigma^* \times \Sigma^*$ and a positive integer k such that the following two conditions are satisfied:

1. For all $x \in \Sigma^*$, $x \in L \Leftrightarrow \exists y \in \Sigma^*$ such that $(x, y) \in R$ and $|y| \in O\left(|x|^k\right)$; and
2. the language $L_R = \{x\#y : (x, y) \in R\}$ over $\Sigma \cup \{\#\}$ is decidable by a Turing machine in polynomial time.

A Turing machine that decides L_R is called a *verifier* for L and a y such that $(x, y) \in R$ is called a *certificate of membership* of x in L.

In general, a verifier does not have to be polynomial-time. However, for L to be in **NP**, there must be a verifier that runs in polynomial time.

Formal definition for NP-completeness

There are many equivalent ways of describing **NP**-completeness.

Let L be a language over a finite alphabet Σ.

L is **NP**-complete if, and only if, the following two conditions are satisfied:

1. $L \in \mathbf{NP}$; and
2. any $L' \in \mathbf{NP}$ is polynomial-time-reducible to L (written as $L' \leq_p L$), where $L' \leq_p L$ if, and only if, the following two conditions are satisfied:
   1. There exists $f : \Sigma^* \to \Sigma^*$ such that $\forall w \in \Sigma^* (w \in L' \Leftrightarrow f(w) \in L)$; and
   2. there exists a polynomial-time Turing machine which halts with $f(w)$ on its tape on any input w.

See also

- Game complexity
- Unsolved problems in computer science
- Unsolved problems in mathematics

Further reading

- A. S. Fraenkel and D. Lichtenstein, Computing a perfect strategy for n*n chess requires time exponential in n, Proc. 8th Int. Coll. *Automata, Languages, and Programming*, Springer LNCS 115 (1981) 278–293 and *J. Comb. Th. A* 31 (1981) 199–214.
- E. Berlekamp and D. Wolfe, Mathematical Go: Chilling Gets the Last Point, A. K. Peters, 1994. D. Wolfe, Go endgames are hard, MSRI Combinatorial Game Theory Research Worksh., 2000.
- Neil Immerman. Languages Which Capture Complexity Classes. *15th ACM STOC Symposium*, pp.347–354. 1983.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein (2001). "Chapter 34: NP-Completeness". *Introduction to Algorithms* (Second ed.). MIT Press and McGraw-Hill. pp. 966–1021. ISBN 0-262-03293-7.
- Markoff, John, "Prizes Aside, the P-NP Puzzler Has Consequences" [21], The New York Times, October 8, 2009
- Christos Papadimitriou (1993). "Chapter 14: On P vs. NP". *Computational Complexity* (1st ed.). Addison Wesley. pp. 329–356. ISBN 0-201-53082-1.
- Lance Fortnow (September 2009), "The Status of the P Versus NP Problem" [22], *Communications of the ACM* **52** (9): pp. 78–86, doi:10.1145/1562164.1562186

External links

- The Clay Mathematics Institute Millennium Prize Problems [23]
- The Clay Math Institute Official Problem Description [24]PDF (118 KB)
- Ian Stewart on Minesweeper as **NP**-complete at The Clay Math Institute [25]
- Gerhard J. Woeginger. The P-versus-NP page [26]. A list of links to a number of purported solutions to the problem. Some of these links state that P equals NP, some of them state the opposite. It is probable that all these alleged solutions are incorrect.
- Computational Complexity of Games and Puzzles [7]
- *Complexity Zoo*: Class P [11], *Complexity Zoo*: Class NP [27]
- Scott Aaronson's Shtetl Optimized blog: Reasons to believe [28], a list of justifications for the belief that P ≠ NP

References

[1] R. E. Ladner "On the structure of polynomial time reducibility," J.ACM, 22, pp. 151–171, 1975. Corollary 1.1. ACM site (http://portal.acm.org/citation.cfm?id=321877&dl=ACM&coll=&CFID=15151515&CFTOKEN=6184618).

[2] Lance Fortnow, *The status of the P versus NP problem* (http://www.cs.uchicago.edu/~fortnow/papers/pnp-cacm.pdf), Communications of the ACM 52 (2009), no. 9, pp. 78–86. doi:10.1145/1562164.1562186

[3] "Millennium Prize problems" (http://www.claymath.org/millennium/). 2000-05-24. . Retrieved 2008-01-12.

[4] Cook, Stephen (April 2000). *The P versus NP Problem* (http://www.claymath.org/millennium/P_vs_NP/Official_Problem_Description.pdf). Clay Mathematics Institute. . Retrieved 2006-10-18.

[5] *Complexity Zoo*: Class FP (http://qwiki.stanford.edu/wiki/Complexity_Zoo:F#fp): "FP = FNP if and only if P = NP".

[6] Sipser, Michael: *Introduction to the Theory of Computation, Second Edition, International Edition*, page 270. Thomson Course Technology, 2006. Definition 7.19 and Theorem 7.20.

[7] William I. Gasarch (June 2002). "The P=?NP poll." (http://www.cs.umd.edu/~gasarch/papers/poll.pdf) (PDF). *SIGACT News* **33** (2): 34–47. doi:10.1145/1052796.1052804. . Retrieved 2008-12-29.

[8] M. Agrawal, N. Kayal, N. Saxena. "Primes is in P" (http://www.cse.iitk.ac.in/users/manindra/algebra/primality_v6.pdf) (PDF). . Retrieved 2008-12-29.

[9] AKS primality test

[10] Scott Aaronson. "PHYS771 Lecture 6: P, NP, and Friends" (http://www.scottaaronson.com/democritus/lec6.html). . Retrieved 2007-08-27.

[11] Aviezri Fraenkel and D. Lichtenstein (1981). "Computing a perfect strategy for n×n chess requires time exponential in n". *J. Comb. Th. A* (31): 199–214.

[12] David Eppstein. "Computational Complexity of Games and Puzzles" (http://www.ics.uci.edu/~eppstein/cgt/hard.html). .

[13] Pisinger, D. 2003. "Where are the hard knapsack problems?" Technical Report 2003/08, Department of Computer Science, University of Copenhagen, Copenhagen, Denmark

[14] Scott Aaronson. "Reasons to believe" (http://scottaaronson.com/blog/?p=122). ., point 9.

[15] Berger B, Leighton T (1998). "Protein folding in the hydrophobic-hydrophilic (HP) model is NP-complete". *J. Comput. Biol.* **5** (1): 27–40. doi:10.1145/1052796.1052804. PMID 9541869.

[16] L. R. Foulds (October 1983). "The Heuristic Problem-Solving Approach" (http://www.jstor.org/pss/2580891). *The Journal of the Operational Research Society* **34** (10): 927–934. doi:10.2307/2580891. .

[17] T. P. Baker, J. Gill, R. Solovay. *Relativizations of the P =? NP Question*. SIAM Journal on Computing, 4(4): 431-442 (1975)

[18] S. Aaronson and A. Wigderson. Algebrization: A New Barrier in Complexity Theory, in Proceedings of ACM STOC'2008, pp. 731-740.

[19] Aaronson, Scott, *Is P Versus NP Formally Independent?* (http://www.scottaaronson.com/papers/pnp.pdf), .

[20] Ben-David, Shai; Halevi, Shai (1992), *On the independence of P versus NP* (http://www.cs.technion.ac.il/~shai/ph.ps.gz), Technical Report, **714**, Technion, .

[21] http://www.nytimes.com/2009/10/08/science/Wpolynom.html?_r=1

[22] http://cacm.acm.org/magazines/2009/9/38904-the-status-of-the-p-versus-np-problem/fulltext

[23] http://www.claymath.org/millennium/

[24] http://www.claymath.org/millennium/P_vs_NP/Official_Problem_Description.pdf

[25] http://www.claymath.org/Popular_Lectures/Minesweeper/

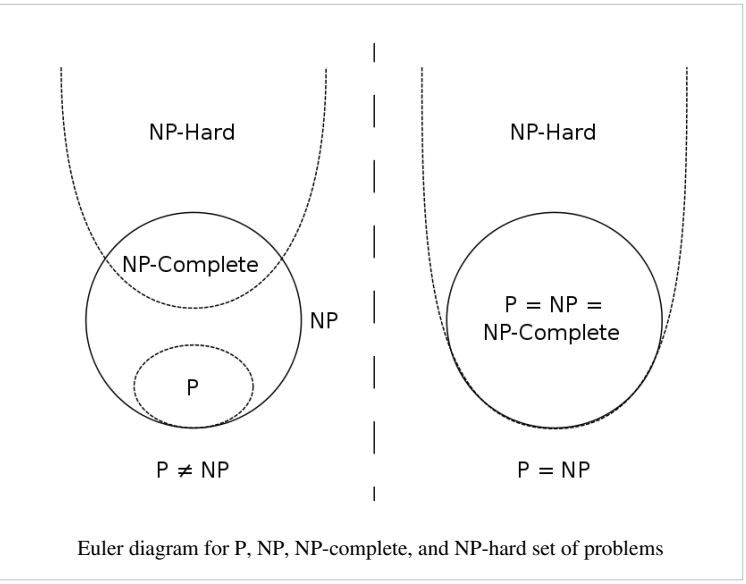
[26] http://www.win.tue.nl/~gwoegi/P-versus-NP.htm

[27] http://qwiki.stanford.edu/wiki/Complexity_Zoo:N#np

[28] http://scottaaronson.com/blog/?p=122

NP-hard

NP-hard (non-deterministic polynomial-time hard), in computational complexity theory, is a class of problems that are, informally, "at least as hard as the hardest problems in NP". A problem *H* is NP-hard if and only if there is an NP-complete problem L that is polynomial time Turing-reducible to H (i.e., $L \leq_T H$). In other words, *L* can be solved in polynomial time by an oracle machine with an oracle for *H*. Informally, we can think of an algorithm that can call such an oracle machine as a subroutine for solving *H*, and solves *L* in polynomial time, if the subroutine call takes only one step to compute. NP-hard problems may be of any type: decision problems, search problems, or optimization problems.



Euler diagram for P, NP, NP-complete, and NP-hard set of problems

As consequences of definition, we have (note that these are claims, not definitions):

- Problem *H* is at least as hard as L, because H can be used to solve *L*;
- Since *L* is NP-complete, and hence the hardest in class NP, also problem *H* is at least as hard as NP, but *H* does not have to be in NP and hence does not have to be a decision problem (even if it is a decision problem, it need not be in NP);
- Since NP-complete problems transform to each other by polynomial-time many-one reduction (also called polynomial transformation), all NP-complete problems can be solved in polynomial time by a reduction to *H*, thus all problems in NP reduce to *H*; note, however, that this involves combining two different transformations: from NP-complete decision problems to NP-complete problem *L* by polynomial transformation, and from *L* to *H* by polynomial Turing reduction;
- If there is a polynomial algorithm for any NP-hard problem, then there are polynomial algorithms for all problems in NP, and hence P = NP;
- If P ≠ NP, then NP-hard problems have no solutions in polynomial time, while P = NP does not resolve whether the NP-hard problems can be solved in polynomial time;
- If an optimization problem H has an NP-complete decision version *L*, then *H* is NP-hard.

A common mistake is to think that the *NP* in *NP-hard* stands for *non-polynomial*. Although it is widely suspected that there are no polynomial-time algorithms for NP-hard problems, this has never been proven. Moreover, the class NP also contains all problems which can be solved in polynomial time.

Examples

An example of an NP-hard problem is the decision subset sum problem, which is this: given a set of integers, does any non-empty subset of them add up to zero? That is a *yes*/*no* question, and happens to be NP-complete. Another example of an NP-hard problem is the optimization problem of finding the least-cost route through all nodes of a weighted graph. This is commonly known as the Traveling Salesman Problem.

There are also decision problems that are NP-hard but not NP-complete, for example the halting problem. This is the problem which asks "given a program and its input, will it run forever?" That's a *yes*/*no* question, so this is a decision problem. It is easy to prove that the halting problem is *NP-hard* but not *NP-complete*. For example, the Boolean satisfiability problem can be reduced to the halting problem by transforming it to the description of a Turing machine that tries all truth value assignments and when it finds one that satisfies the formula it halts and otherwise it goes into an infinite loop. It is also easy to see that the halting problem is not in *NP* since all problems in NP are decidable in a finite number of operations, while the halting problem, in general, is not. There are also NP-hard problems that are neither NP-complete nor undecidable. For instance, the language of True quantified Boolean formulas is decidable in polynomial space, but not non-deterministic polynomial time (unless NP = PSPACE).

Alternative definitions

An alternative definition of NP-hard that is often used restricts NP-hard to decision problems and then uses polynomial-time many-one reduction instead of Turing reduction. So, formally, a language L is *NP-hard* if $\forall L' \in \text{NP}, L' \leq L$. If it is also the case that L is in *NP*, then L is called *NP-complete*.

NP-naming convention

The NP-family naming system is confusing: NP-hard problems are not all NP, despite having *NP* as the prefix of their class name. However, the names are now entrenched and unlikely to change. On the other hand, the *NP*-naming system has some deeper sense, because the NP family is defined in relation to the class NP:

NP-hard

: As hard as the hardest problems in NP. Such problems need not be in NP; indeed, they may not even be decision problems.

NP-complete

: These are the hardest problems in NP. Such a problem is NP-hard and in NP.

NP-easy

: At most as hard as NP, but not necessarily in NP, since they may not be decision problems.

NP-equivalent

: Exactly as difficult as the hardest problems in NP, but not necessarily in NP.

References

- [1] Introduction to NP-problems
- Michael R. Garey and David S. Johnson (1979). [2] *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman. ISBN 0-7167-1045-5.

References

[1] http://explorations.chasrmartin.com/2008/11/24/what-are-p-np-complete-and-np-hard/

[2] http://www.amazon.com/dp/0716710455

Computational complexity theory

Computational complexity theory is a branch of the theory of computation in computer science and mathematics that focuses on classifying computational problems according to their inherent difficulty. In this context, a computational problem is understood to be a task that is in principle amenable to being solved by a computer. Informally, a computational problem consists of problem instances and solutions to these problem instances. For example, primality testing is the problem of determining whether a given number is prime or not. The instances of this problem are natural numbers, and the solution to an instance is *yes* or *no* based on whether the number is prime or not.

A problem is regarded as inherently difficult if solving the problem requires a large amount of resources, whatever the algorithm used for solving it. The theory formalizes this intuition, by introducing mathematical models of computation to study these problems and quantifying the amount of resources needed to solve them, such as time and storage. Other complexity measures are also used, such as the amount of communication (used in communication complexity), the number of gates in a circuit (used in circuit complexity) and the number of processors (used in parallel computing). In particular, computational complexity theory determines the practical limits on what computers can and cannot do.

Closely related fields in theoretical computer science are analysis of algorithms and computability theory. A key distinction between computational complexity theory and analysis of algorithms is that the latter is devoted to analyzing the amount of resources needed by a particular algorithm to solve a problem, whereas the former asks a more general question about all possible algorithms that could be used to solve the same problem. More precisely, it tries to classify problems that can or cannot be solved with appropriately restricted resources. In turn, imposing restrictions on the available resources is what distinguishes computational complexity from computability theory: the latter theory asks what kind of problems can be solved in principle algorithmically.

Computational problems

Problem instances

A computational problem can be viewed as an infinite collection of *instances* together with a *solution* for every instance. The input string for a computational problem is referred to as a problem instance, and should not be confused with the problem itself. In computational complexity theory, a problem refers to the abstract question to be solved. In contrast, an instance of this problem is a rather concrete utterance, which can serve as the input for a decision problem. For example, consider the problem of primality testing. The instance is a number and the solution is "yes" if the number is prime and "no" otherwise. Alternately, the instance is a particular input to the problem, and the solution is the output corresponding to the given input.

An optimal traveling salesperson tour through Germany's 15 largest cities. It is the shortest among 43 589 145 600[1] possible tours visiting each city exactly once.

To further highlight the difference between a problem and an instance, consider the following instance of the decision version of the traveling salesman problem: Is there a route of length at most 2000 kilometres passing through all of Germany's 15 largest cities? The answer to this particular problem instance is of little use for solving other instances of the problem, such as asking for a round trip through all sights in Milan whose total length is at most 10 km. For this reason, complexity theory addresses computational problems and not particular problem instances.

Representing problem instances

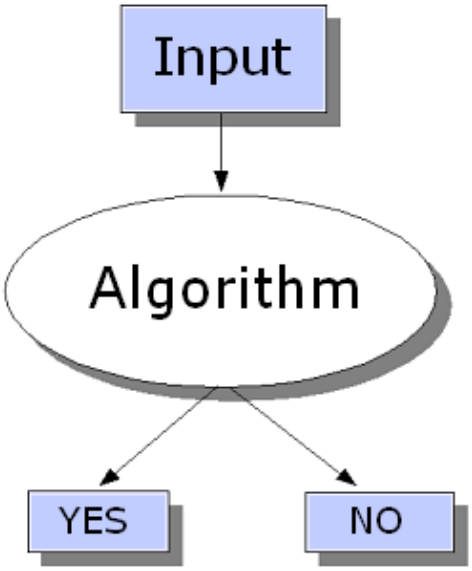
When considering computational problems, a problem instance is a string over an alphabet. Usually, the alphabet is taken to be the binary alphabet (i.e., the set {0,1}), and thus the strings are bitstrings. As in a real-world computer, mathematical objects other than bitstrings must be suitably encoded. For example, integers can be represented in binary notation, and graphs can be encoded directly via their adjacency matrices, or via encoding their adjacency lists in binary.

Even though some proofs of complexity-theoretic theorems regularly assume some concrete choice of input encoding, one tries to keep the discussion abstract enough to be independent of the choice of encoding. This can be achieved by ensuring that different representations can be transformed into each other efficiently.

Decision problems as formal languages

Decision problems are one of the central objects of study in computational complexity theory. A decision problem is a special type of computational problem whose answer is either *yes* or *no*, or alternately either 1 or 0. A decision problem can be viewed as a formal language, where the members of the language are instances whose answer is yes, and the non-members are those instances whose output is no. The objective is to decide, with the aid of an algorithm, whether a given input string is member of the formal language under consideration. If the algorithm deciding this problem returns the answer *yes*, the algorithm is said to accept the input string, otherwise it is said to reject the input.

An example of a decision problem is the following. The input is an arbitrary graph. The problem consists in deciding whether the given graph is connected, or not. The formal language associated with this decision problem is then the set of all connected graphs—of course, to obtain a precise definition of this language, one has to decide how graphs are encoded as binary strings.



A decision problem has only two possible outputs, *yes* or *no* (or alternately 1 or 0) on any input.

Function problems

A function problem is a computational problem where a single output (of a total function) is expected for every input, but the output is more complex than that of a decision problem, that is, it isn't just yes or no. Notable examples include the traveling salesman problem and the integer factorization problem.

It is tempting to think that the notion of function problems is much richer than the notion of decision problems. However, this is not really the case, since function problems can be recast as decision problems. For example, the multiplication of two integers can be expressed as the set of triples (a, b, c) such that the relation $a \times b = c$ holds. Deciding whether a given triple is member of this set corresponds to solving the problem of multiplying two numbers.

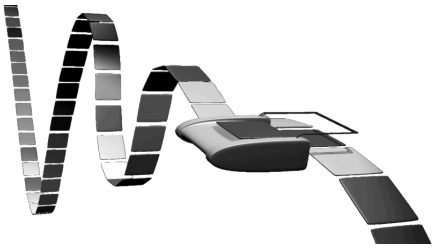
Measuring the size of an instance

To measure the difficulty of solving a computational problem, one may wish to see how much time the best algorithm requires to solve the problem. However, the running time may, in general, depend on the instance. In particular, larger instances will require more time to solve. Thus the time required to solve a problem (or the space required, or any measure of complexity) is calculated as function of the size of the instance. This is usually taken to be the size of the input in bits. Complexity theory is interested in how algorithms scale with an increase in the input size. For instance, in the problem of finding whether a graph is connected, how much more time does it take to solve a problem for a graph with $2n$ vertices compared to the time taken for a graph with n vertices?

If the input size is n, the time taken can be expressed as a function of n. Since the time taken on different inputs of the same size can be different, the worst-case time complexity $T(n)$ is defined to be the maximum time taken over all inputs of size n. If $T(n)$ is a polynomial in n, then the algorithm is said to be a polynomial time algorithm. Cobham's thesis says that a problem can be solved with a feasible amount of resources if it admits a polynomial time algorithm.

Machine models and complexity measures

Turing Machine



An artistic representation of a Turing machine

A Turing machine is a mathematical model of a general computing machine. It is a theoretical device that manipulates symbols contained on a strip of tape. Turing machines are not intended as a practical computing technology, but rather as a thought experiment representing a computing machine. It is believed that if a problem can be solved by an algorithm, there exists a Turing machine which solves the problem. Indeed, this is the statement of the Church–Turing thesis. Furthermore, it is known that everything that can be computed on other models of computation known to us today, such as a RAM machine, Conway's Game of Life, cellular automata or any programming language can be computed on a Turing machine. Since Turing machines are easy to analyze mathematically, and are believed to be as powerful as any other model of computation, the Turing machine is the most commonly used model in complexity theory.

Many types of Turing machines are used to define complexity classes, such as deterministic Turing machines, probabilistic Turing machines, non-deterministic Turing machines, quantum Turing machines, symmetric Turing machines and alternating Turing machines. They are all equally powerful in principle, but when resources (such as time or space) are bounded, some of these may be more powerful than others.

A deterministic Turing machine is the most basic Turing machine, which uses a fixed set of rules to determine its future actions. A probabilistic Turing machine is a deterministic Turing machine with an extra supply of random bits. The ability to make probabilistic decisions often helps algorithms solve problems more efficiently. Algorithms which use random bits are called randomized algorithms. A non-deterministic Turing machine is a deterministic Turing machine with an added feature of non-determinism, which allows a Turing machine to have multiple possible future actions from a given state. One way to view non-determinism is that the Turing machine branches into many possible computational paths at each step, and if it solves the problem in any of these branches, it is said to have solved the problem. Clearly, this model is not meant to be a physically realizable model, it is just a theoretically interesting abstract machine which gives rise to particularly interesting complexity classes. For examples, see nondeterministic algorithm.

Other machine models

Many machine models different from the standard multitape Turing machines have been proposed in the literature, for example random access machines. Perhaps surprisingly, each of these models can be converted to another without substantial overhead in time and memory consumption.[2] What all these models have in common is that the machines operate deterministically.

However, some computational problems are easier to analyze in terms of more unusual resources. For example, a nondeterministic Turing machine is a computational model that is allowed to branch out to check many different possibilities at once. The nondeterministic Turing machine has very little to do with how we physically want to compute algorithms, but its branching exactly captures many of the mathematical models we want to analyze, so that nondeterministic time is a very important resource in analyzing computational problems.

Complexity measures

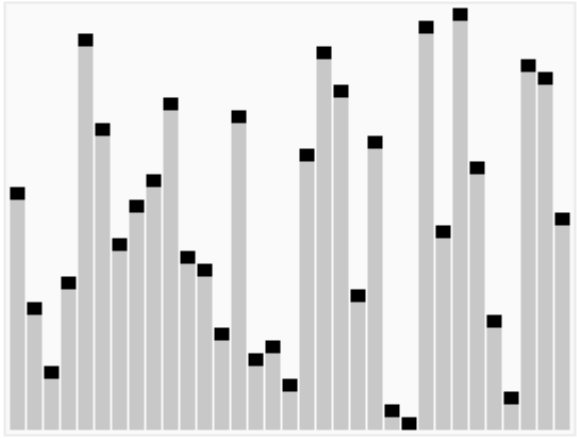
For a precise definition of what it means to solve a problem using a given amount of time and space, a computational model such as the deterministic Turing machine is used. The *time required* by a deterministic Turing machine M on input x is the total number of state transitions, or steps, the machine makes before it halts and outputs the answer ("yes" or "no"). A Turing machine M is said to operate within time $f(n)$, if the time required by M on each input of length n is at most $f(n)$. A decision problem A can be solved in time $f(n)$ if there exists a Turing machine operating in time $f(n)$ which solves the problem. Since complexity theory is interested in classifying problems based on their difficulty, one defines sets of problems based on some criteria. For instance, the set of problems solvable within time $f(n)$ on a deterministic Turing machine is then denoted by DTIME($f(n)$).

Analogous definitions can be made for space requirements. Although time and space are the most well-known complexity resources, any complexity measure can be viewed as a computational resource. Complexity measures are very generally defined by the Blum complexity axioms. Other complexity measures used in complexity theory include communication complexity, circuit complexity, and decision tree complexity.

Best, worst and average case complexity

The best, worst and average case complexity refer to three different ways of measuring the time complexity (or any other complexity measure) of different inputs of the same size. Since some inputs of size n may be faster to solve than others, we define the following complexities:

- Best-case complexity: This is the complexity of solving the problem for the best input of size n.
- Worst-case complexity: This is the complexity of solving the problem for the worst input of size n.
- Average-case complexity: This is the complexity of solving the problem on an average. This complexity is only defined with respect to a probability distribution over the inputs. For instance, if all inputs of the same size are assumed to be equally likely to appear, the average case complexity can be defined with respect to the uniform distribution over all inputs of size n.



Visualization of the quicksort algorithm that has average case performance $\Theta(n \log n)$.

For example, consider the deterministic sorting algorithm quicksort. This solves the problem of sorting a list of integers which is given as the input. The best-case scenario is when the input is already sorted, and the algorithm takes time O(n log n) for such inputs. The worst-case is when the input is sorted in reverse order, and the algorithm takes time O(n^2) for this case. If we assume that all possible permutations of the input list are equally likely, the average time taken for sorting is O(n log n).

Upper and lower bounds on the complexity of problems

To classify the computation time (or similar resources, such as space consumption), one is interested in proving upper and lower bounds on the minimum amount of time required by the most efficient algorithm solving a given problem. The complexity of an algorithm is usually taken to be its worst-case complexity, unless specified otherwise. Analyzing a particular algorithm falls under the field of analysis of algorithms. To show an upper bound $T(n)$ on the time complexity of a problem, one needs to show only that there is a particular algorithm with running time at most $T(n)$. However, proving lower bounds is much more difficult, since lower bounds make a statement about all possible algorithms that solve a given problem. The phrase "all possible algorithms" includes not just the algorithms

known today, but any algorithm that might be discovered in the future. To show a lower bound of $T(n)$ for a problem requires showing that no algorithm can have time complexity lower than $T(n)$.

Upper and lower bounds are usually stated using the big Oh notation, which hides constant factors and smaller terms. This makes the bounds independent of the specific details of the computational model used. For instance, if $T(n) = 7n^2 + 15n + 40$, in big Oh notation one would write $T(n) = \mathrm{O}(n^2)$.

Complexity classes

Defining complexity classes

A **complexity class** is a set of problems of related complexity. Simpler complexity classes are defined by the following factors:

- The type of computational problem: The most commonly used problems are decision problems. However, complexity classes can be defined based on function problems, counting problems, optimization problems, promise problems, etc.
- The model of computation: The most common model of computation is the deterministic Turing machine, but many complexity classes are based on nondeterministic Turing machines, boolean circuits, quantum Turing machines, monotone circuits, etc.
- The resources (or resources) that are being bounded and the bounds: These two properties are usually stated together, such as "polynomial time", "logarithmic space", "constant depth", etc.

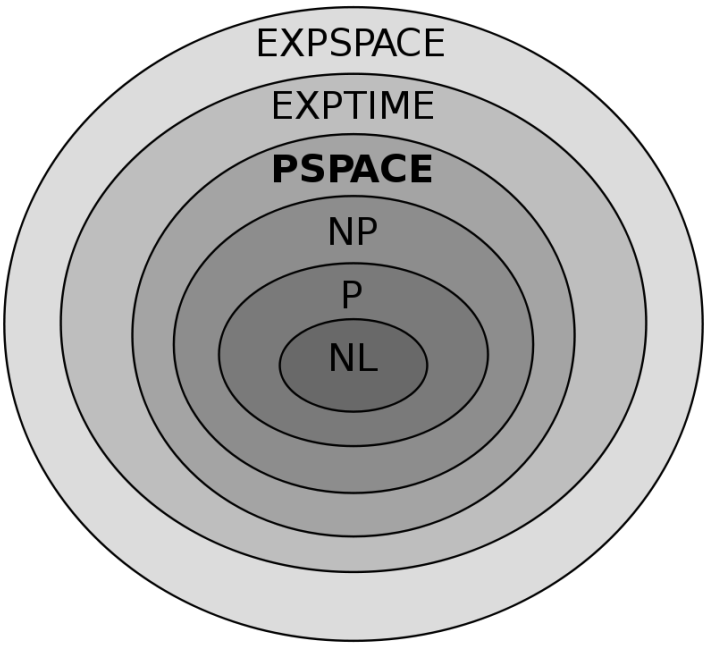
Of course, some complexity classes have complex definitions which do not fit into this framework. Thus, a typical complexity class has a definition like the following:

> The set of decision problems solvable by a deterministic Turing machine within time $f(n)$. (This complexity class is known as DTIME($f(n)$).)

But bounding the computation time above by some concrete function $f(n)$ often yields complexity classes that depend on the chosen machine model. For instance, the language $\{xx \mid x \text{ is any binary string}\}$ can be solved in linear time on a multi-tape Turing machine, but necessarily requires quadratic time in the model of single-tape Turing machines. If we allow polynomial variations in running time, Cobham-Edmonds thesis states that "the time complexities in any two reasonable and general models of computation are polynomially related" (Goldreich 2008, Chapter 1.2). This forms the basis for the complexity class P, which is the set of decision problems solvable by a deterministic Turing machine within polynomial time. The corresponding set of function problems is FP.

Important complexity classes

Many important complexity classes can be defined by bounding the time or space used by the algorithm. Some important complexity classes of decision problems defined in this manner are the following:



A representation of the relation among complexity classes

Complexity class	Model of computation	**Resource constraint**
DTIME($f(n)$)	Deterministic Turing machine	Time $f(n)$
P	Deterministic Turing machine	Time poly(n)
EXPTIME	Deterministic Turing machine	Time $2^{\text{poly}(n)}$
NTIME($f(n)$)	Non-deterministic Turing machine	Time $f(n)$
NP	Non-deterministic Turing machine	Time poly(n)
NEXPTIME	Non-deterministic Turing machine	Time $2^{\text{poly}(n)}$
DSPACE($f(n)$)	Deterministic Turing machine	Space $f(n)$
L	Deterministic Turing machine	Space O(log n)
PSPACE	Deterministic Turing machine	Space poly(n)
EXPSPACE	Deterministic Turing machine	Space $2^{\text{poly}(n)}$
NSPACE($f(n)$)	Non-deterministic Turing machine	Space $f(n)$
NL	Non-deterministic Turing machine	Space O(log n)
NPSPACE	Non-deterministic Turing machine	Space poly(n)
NEXPSPACE	Non-deterministic Turing machine	Space $2^{\text{poly}(n)}$

It turns out that PSPACE = NPSPACE and EXPSPACE = NEXPSPACE by Savitch's theorem.

Other important complexity classes include BPP, ZPP and RP, which are defined using probabilistic Turing machines; AC and NC, which are defined using boolean circuits and BQP and QMA, which are defined using quantum Turing machines. #P is an important complexity class of counting problems (not decision problems). Classes like IP and AM are defined using Interactive proof systems. ALL is the class of all decision problems.

Hierarchy theorems

For the complexity classes defined in this way, it is desirable to prove that relaxing the requirements on (say) computation time indeed defines a bigger set of problems. In particular, although DTIME(n) is contained in DTIME(n^2), it would be interesting to know if the inclusion is strict. For time and space requirements, the answer to such questions is given by the time and space hierarchy theorems respectively. They are called hierarchy theorems because they induce a proper hierarchy on the classes defined by constraining the respective resources. Thus there are pairs of complexity classes such that one is properly included in the other. Having deduced such proper set inclusions, we can proceed to make quantitative statements about how much more additional time or space is needed in order to increase the number of problems that can be solved.

More precisely, the time hierarchy theorem states that

$$\mathrm{DTIME}\big(f(n)\big) \subsetneq \mathrm{DTIME}\big(f(n)\cdot\log^2(f(n))\big).$$

The space hierarchy theorem states that

$$\mathrm{DSPACE}\big(f(n)\big) \subsetneq \mathrm{DSPACE}\big(f(n)\cdot\log(f(n))\big).$$

The time and space hierarchy theorems form the basis for most separation results of complexity classes. For instance, the time hierarchy theorem tells us that P is strictly contained in EXPTIME, and the space hierarchy theorem tells us that L is strictly contained in PSPACE.

Reduction

Many complexity classes are defined using the concept of a reduction. A reduction is a transformation of one problem into another problem. It captures the informal notion of a problem being at least as difficult as another problem. For instance, if a problem X can be solved using an algorithm for Y, X is no more difficult than Y, and we say that X *reduces* to Y. There are many different type of reductions, based on the method of reduction, such as Cook reductions, Karp reductions and Levin reductions, and the bound on the complexity of reductions, such as polynomial-time reductions or log-space reductions.

The most commonly used reduction is a polynomial-time reduction. This means that the reduction process takes polynomial time. For example, the problem of squaring an integer can be reduced to the problem of multiplying two integers. This means an algorithm for multiplying two integers can be used to square an integer. Indeed, this can be done by giving the same input to both inputs of the multiplication algorithm. Thus we see that squaring is not more difficult than multiplication, since squaring can be reduced to multiplication.

This motivates the concept of a problem being hard for a complexity class. A problem X is *hard* for a class of problems C if every problem in C can be reduced to X. Thus no problem in C is harder than X, since an algorithm for X allows us to solve any problem in C. Of course, the notion of hard problems depends on the type of reduction being used. For complexity classes larger than P, polynomial-time reductions are commonly used. In particular, the set of problems that are hard for NP is the set of NP-hard problems.

If a problem X is in C and hard for C, then X is said to be *complete* for C. This means that X is the hardest problem in C (Since many problems could be equally hard, one might say that X is one of the hardest problems in C). Thus the class of NP-complete problems contains the most difficult problems in NP, in the sense that they are the ones most likely not to be in P. Because the problem P = NP is not solved, being able to reduce another problem, Π_1, to a known NP-complete problem, Π_2, would indicate that there is no known polynomial-time solution for Π_1. This is due to the fact that a polynomial-time solution to Π_1 would yield a polynomial-time solution to Π_2. Similarly, because all NP problems can be reduced to the set, finding an NP-complete problem that can be solved in polynomial time would mean that P = NP.[3]

Important open problems

P versus NP problem

The complexity class P is often seen as a mathematical abstraction modeling those computational tasks that admit an efficient algorithm. This hypothesis is called the Cobham–Edmonds thesis. The complexity class NP, on the other hand, contains many problems that people would like to solve efficiently, but for which no efficient algorithm is known, such as the Boolean satisfiability problem, the Hamiltonian path problem and the vertex cover problem. Since deterministic Turing machines are special nondeterministic Turing machines, it is easily observed that each problem in P is also member of the class NP.

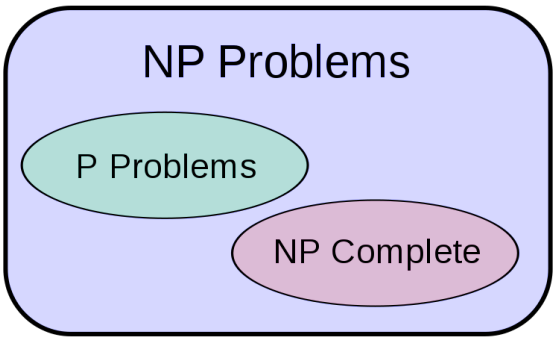


Diagram of complexity classes provided that P ≠ NP. The existence of problems in NP outside both P and NP-complete in this case was established by Ladner.[4]

The question of whether P equals NP is one of the most important open questions in theoretical computer science because of the wide implications of a solution.[3] If the answer is yes, many important problems can be shown to have more efficient solutions. These include various types of integer programming problems in operations research, many problems in logistics, protein structure prediction in biology,[5] and the ability to find formal proofs of pure mathematics theorems.[6] The P versus NP problem is one of the Millennium Prize Problems proposed by the Clay Mathematics Institute. There is a US$1,000,000 prize for resolving the problem.[7]

Problems in NP not known to be in P or NP-complete

It was shown by Ladner that if P ≠ NP then there exist problems in NP that are neither in P nor NP-complete.[4] Such problems are called NP-intermediate problems. The graph isomorphism problem, the discrete logarithm problem and the integer factorization problem are examples of problems believed to be NP-intermediate. They are some of the very few NP problems not known to be in P or to be NP-complete.

The graph isomorphism problem is the computational problem of determining whether two finite graphs are isomorphic. An important unsolved problem in complexity theory is whether the graph isomorphism problem is in P, NP-complete, or NP-intermediate. The answer is not known, but it is believed that the problem is at least not NP-complete.[8] If graph isomorphism is NP-complete, the polynomial time hierarchy collapses to its second level.[9] Since it is widely believed that the polynomial hierarchy does not collapse to any finite level, it is believed that graph isomorphism is not NP-complete. The best algorithm for this problem, due to Laszlo Babai and Eugene Luks has run time $2^{O(\sqrt{(n \log n)})}$ for graphs with n vertices.

The integer factorization problem is the computational problem of determining the prime factorization of a given integer. Phrased as a decision problem, it is the problem of deciding whether the input has a factor less than k. No efficient integer factorization algorithm is known, and this fact forms the basis of several modern cryptographic systems, such as the RSA algorithm. The integer factorization problem is in NP and in co-NP (and even in UP and co-UP[10]). If the problem is NP-complete, the polynomial time hierarchy will collapse to its first level (i.e., NP will equal co-NP). The best known algorithm for integer factorization is the general number field sieve, which takes time $O(e^{(64/9)}1/3(n.\log 2)^{1/3}(\log (n.\log 2))^{2/3})$ to factor an n-bit integer. However, the best known quantum algorithm for this problem, Shor's algorithm, does run in polynomial time. Unfortunately, this fact doesn't say much about where the problem lies with respect to non-quantum complexity classes.

Separations between other complexity classes

Many known complexity classes are suspected to be unequal, but this has not been proved. For instance P ⊆ NP ⊆ PP ⊆ PSPACE, but it is possible that P = PSPACE. If P is not equal to NP, then P is not equal to PSPACE either. Since there are many known complexity classes between P and PSPACE, such as RP, BPP, PP, BQP, MA, PH, etc., it is possible that all these complexity classes collapse to one class. Proving that any of these classes are unequal would be a major breakthrough in complexity theory.

Along the same lines, co-NP is the class containing the complement problems (i.e. problems with the *yes*/*no* answers reversed) of NP problems. It is believed[11] that NP is not equal to co-NP, however, it has not yet been proven. It has been shown that if these two complexity classes are not equal then P is not equal to NP.

Similarly, it is not known if L (the set of all problems that can be solved in logarithmic space) is strictly contained in P or equal to P. Again, there are many complexity classes between the two, such as NL and NC, and it is not known if they are distinct or equal classes.

Intractability

Problems that can be solved but not fast enough for the solution to be useful are called *intractable*.[12] In complexity theory, problems that lack polynomial-time solutions are considered to be intractable for more than the smallest inputs. In fact, the Cobham–Edmonds thesis states that only those problems that can be solved in polynomial time can be feasibly computed on some computational device. Problems that are known to be intractable in this sense include those that are EXPTIME-hard. If NP is not the same as P, then the NP-complete problems are also intractable in this sense. To see why exponential-time algorithms might be unusable in practice, consider a program that makes 2^n operations before halting. For small n, say 100, and assuming for the sake of example that the computer does 10^{12} operations each second, the program would run for about 4×10^{10} years, which is roughly the age of the universe. Even with a much faster computer, the program would only be useful for very small instances and in that sense the intractability of a problem is somewhat independent of technological progress. Nevertheless a polynomial time algorithm is not always practical. If its running time is, say, n^{15}, it is unreasonable to consider it efficient and it is still useless except on small instances.

What intractability means in practice is open to debate. Saying that a problem is not in P does not imply that all large cases of the problem are hard or even that most of them are. For example the decision problem in Presburger arithmetic has been shown not to be in P, yet algorithms have been written that solve the problem in reasonable times in most cases. Similarly, algorithms can solve the NP-complete knapsack problem over a wide range of sizes in less than quadratic time and SAT solvers routinely handle large instances of the NP-complete Boolean satisfiability problem.

Continuous complexity theory

Continuous complexity theory can refer to complexity theory of problems that involve continuous functions that are approximated by discretizations, as studied in numerical analysis. One approach to complexity theory of numerical analysis[13] is information based complexity.

Continuous complexity theory can also refer to complexity theory of the use of analog computation, which uses continuous dynamical systems and differential equations.[14] Control theory can be considered a form of computation and differential equations are used in the modelling of continuous-time and hybrid discrete-continuous-time systems.[15]

History

Before the actual research explicitly devoted to the complexity of algorithmic problems started off, numerous foundations were laid out by various researchers. Most influential among these was the definition of Turing machines by Alan Turing in 1936, which turned out to be a very robust and flexible notion of computer.

Fortnow and Homer (2003) date the beginning of systematic studies in computational complexity to the seminal paper "On the Computational Complexity of Algorithms" by Juris Hartmanis and Richard Stearns (1965), which laid out the definitions of time and space complexity and proved the hierarchy theorems.

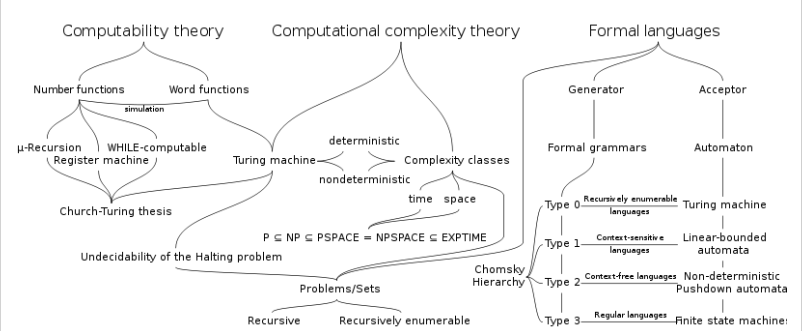
According to Fortnow and Homer (2003), earlier papers studying problems solvable by Turing machines with specific bounded resources include John Myhill's definition of linear bounded automata (Myhill 1960), Raymond Smullyan's study of rudimentary sets (1961), as well as Hisao Yamada's paper on real-time computations (1962). Somewhat earlier, Trakhtenbrot (1956), a pioneer in the field from the USSR, studied another specific complexity measure.[16] As he remembers:

> "However, [my] initial interest [in automata theory] was increasingly set aside in favor of computational complexity, an exciting fusion of combinatorial methods, inherited from switching theory, with the conceptual arsenal of the theory of algorithms. These ideas had occurred to me earlier in 1955 when I coined the term "signalizing function", which is nowadays commonly known as "complexity measure".
>
> —Boris Trakhtenbrot, From Logic to Theoretical Computer Science – An Update. In: Pillars of Computer Science, LNCS 4800, Springer 2008.

In 1967, Manuel Blum developed an axiomatic complexity theory based on his axioms and proved an important result, the so called, speed-up theorem. The field really began to flourish when the US researcher Stephen Cook and, working independently, Leonid Levin in the USSR, proved that there exist practically relevant problems that are NP-complete. In 1972, Richard Karp took this idea a leap forward with his landmark paper, "Reducibility Among Combinatorial Problems", in which he showed that 21 diverse combinatorial and graph theoretical problems, each infamous for its computational intractability, are NP-complete.[17]

See also

- List of computability and complexity topics
- List of important publications in theoretical computer science
- Unsolved problems in computer science
- Category:Computational problems
- List of complexity classes
- Structural complexity theory
- Descriptive complexity theory
- Quantum complexity theory
- Context of computational complexity
- Parameterized Complexity
- Game complexity



Relationship between computability theory, complexity theory and formal language theory.

References

[1] Take one city, and take all possible orders of the other 14 cities. Then divide by two because it does not matter in which direction in time they come after each other: 14!/2 = 43 589 145 600.

[2] See Arora & Barak 2009, Chapter 1: The computational model and why it doesn't matter

[3] See Sipser 2006, Chapter 7: Time complexity

[4] Ladner, Richard E. (1975). "On the structure of polynomial time reducibility" (http://delivery.acm.org/10.1145/330000/321877/p155-ladner.pdf?key1=321877&key2=7146531911&coll=&dl=ACM&CFID=15151515&CFTOKEN=6184618) (PDF). *Journal of the ACM (JACM)* **22** (1): 151–171. doi:10.1145/321864.321877. .

[5] Berger, Bonnie A.; Leighton, T (1998). "Protein folding in the hydrophobic-hydrophilic (HP) model is NP-complete". *Journal of Computational Biology* **5** (1): p27–40. doi:10.1089/cmb.1998.5.27. PMID 9541869.

[6] Cook, Stephen (April 2000). *The P versus NP Problem* (http://www.claymath.org/millennium/P_vs_NP/Official_Problem_Description.pdf). Clay Mathematics Institute. . Retrieved 2006-10-18.

[7] Jaffe, Arthur M. (2006). "The Millennium Grand Challenge in Mathematics" (http://www.ams.org/notices/200606/fea-jaffe.pdf). *Notices of the AMS* **53** (6). . Retrieved 2006-10-18.

[8] Arvind, Vikraman; Kurur, Piyush P. (2006). "Graph isomorphism is in SPP". *Information and Computation* **204** (5): 835–852. doi:10.1016/j.ic.2006.02.002.

[9] Uwe Schöning, "Graph isomorphism is in the low hierarchy", Proceedings of the 4th Annual Symposium on Theoretical Aspects of Computer Science, 1987, 114–124; also: *Journal of Computer and System Sciences*, vol. 37 (1988), 312–323

[10] Lance Fortnow. Computational Complexity Blog: Complexity Class of the Week: Factoring. September 13, 2002. http://weblog.fortnow.com/2002/09/complexity-class-of-week-factoring.html

[11] Boaz Barak's course on Computational Complexity (http://www.cs.princeton.edu/courses/archive/spr06/cos522/) Lecture 2 (http://www.cs.princeton.edu/courses/archive/spr06/cos522/lec2.pdf)

[12] Hopcroft, J.E., Motwani, R. and Ullman, J.D. (2007) Introduction to Automata Theory, Languages, and Computation, Addison Wesley, Boston/San Francisco/New York (page 368)

[13] Complexity Theory and Numerical Analysis (http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.33.4678&rep=rep1&type=pdf), Steve Smale, Acta Numerica, 1997 - Cambridge Univ Press

[14] A Survey on Continuous Time Computations (http://arxiv.org/abs/arxiv:0907.3117), Olivier Bournez, Manuel Campagnolo, New Computational Paradigms. Changing Conceptions of What is Computable. (Cooper, S.B. and L{\"o}we, B. and Sorbi, A., Eds.). New York, Springer-Verlag, pages 383-423. 2008

[15] Computational Techniques for the Verification of Hybrid Systems (http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.70.4296&rep=rep1&type=pdf), Claire J. Tomlin, Ian Mitchell, Alexandre M. Bayen, Meeko Oishi, Proceedings of the IEEE, Vol. 91, No. 7, July 2003.

[16] Trakhtenbrot, B.A.: Signalizing functions and tabular operators. Uchionnye Zapiski Penzenskogo Pedinstituta (Transactions of the Penza Pedagogoical Institute) 4, 75–87 (1956) (in Russian)

[17] Richard M. Karp (1972). " Reducibility Among Combinatorial Problems (http://www.cs.berkeley.edu/~luca/cs172/karp.pdf)". in R. E. Miller and J. W. Thatcher (editors). *Complexity of Computer Computations*. New York: Plenum. pp. 85–103.

Textbooks

- Arora, Sanjeev; Barak, Boaz (2009). *Computational Complexity: A Modern Approach* (http://www.cs.princeton.edu/theory/complexity/). Cambridge. ISBN 978-0-521-42426-4.
- Downey, Rod; Fellows, M. (1999). *Parameterized complexity* (http://www.springer.com/sgw/cda/frontpage/0,11855,5-0-22-1519914-0,00.html?referer=www.springer.de/cgi-bin/search_book.pl?isbn=0-387-94883-X). Berlin, New York: Springer-Verlag.
- Du, Ding-Zhu; Ko, Ker-I (2000). *Theory of Computational Complexity*. John Wiley & Sons. ISBN 978-0-471-34506-0.
- Goldreich, Oded (2008). *Computational Complexity: A Conceptual Perspective* (http://www.wisdom.weizmann.ac.il/~oded/cc-book.html). Cambridge University Press.
- edited by Jan van Leeuwen. Vol.A, Algorithms and complexity. (1990). van Leeuwen, Jan. ed. *Handbook of theoretical computer science (vol. A): algorithms and complexity*. MIT Press. ISBN 978-0-444-88071-0.
- Papadimitriou, Christos (1994). *Computational Complexity* (1st ed.). Addison Wesley. ISBN 0201530821.
- Sipser, Michael (2006). *Introduction to the Theory of Computation* (2nd ed.). USA: Thomson Course Technology. ISBN 0534950973.
- Garey, Michael R.; Johnson, David S. (1979), *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, ISBN 0-7167-1045-5

Surveys

- Cook, Stephen (1983). "An overview of computational complexity". *Commun. ACM* (ACM) **26** (6): 400–408. doi:10.1145/358141.358144. ISSN 0001-0782.
- Fortnow, Lance; Homer, Steven (2003). "A Short History of Computational Complexity" (http://people.cs.uchicago.edu/~fortnow/papers/history.pdf). *Bulletin of the EATCS* **80**: 95–133.
- Mertens, Stephan (2002). "Computational Complexity for Physicists". *Computing in Science and Engg.* (Piscataway, NJ, USA: IEEE Educational Activities Department) **4** (3): 31–47. doi:10.1109/5992.998639. arXiv:cond-mat/0012185. ISSN 1521-9615.

External links

- The Complexity Zoo (http://qwiki.stanford.edu/wiki/Complexity_Zoo)

Transportation theory

In mathematics and economics, **transportation theory** is a name given to the study of optimal transportation and allocation of resources. The problem was formalized by the French mathematician Gaspard Monge in 1781[1] . Major advances were made in the field during World War II by the Soviet/Russian mathematician and economist Leonid Kantorovich[2] . Consequently, the problem as it is now stated is sometimes known as the **Monge–Kantorovich transportation problem**.

Motivation

Mines and factories

Suppose that we have a collection of *n* mines mining iron ore, and a collection of *n* factories which consume the iron ore that the mines produce. Suppose for the sake of argument that these mines and factories form two disjoint subsets *M* and *F* of the Euclidean plane $\mathbf{R}^2$. Suppose also that we have a *cost function* $c : \mathbf{R}^2 \times \mathbf{R}^2 \to [0, \infty]$, so that $c(x, y)$ is the cost of transporting one shipment of iron from x to y. For simplicity, we ignore the time taken to do the transporting. We also assume that each mine can supply only one factory (no splitting of shipments) and that each factory requires precisely one shipment to be in operation (factories cannot work at half- or double-capacity).

Having made the above assumptions, a *transport plan* is a bijection $T : M \to F$ — i.e. an arrangement whereby each mine $m \in M$ supplies precisely one factory $T(m) \in F$. We wish to find the *optimal transport plan*, the plan T whose *total cost*

$$c(T) := \sum_{m \in M} c(m, T(m))$$

is the least of all possible transport plans from M to F.

Moving books: the importance of the cost function

The following simple example illustrates the importance of the cost function in determining the optimal transport plan. Suppose that we have *n* books of equal width on a shelf (the real line), arranged in a single contiguous block. We wish to rearrange them into another contiguous block, but shifted one book-width to the right. Two obvious candidates for the optimal transport plan present themselves:

1. move all *n* books one book-width to the right; ("many small moves")
2. move the left-most book *n* book-widths to the right and leave all other books fixed. ("one big move")

If the cost function is proportional to Euclidean distance ($c(x, y) = \alpha\ |x - y|$) then these two candidates are *both* optimal. If, on the other hand, we choose the strictly convex cost function proportional to the square of Euclidean distance ($c(x, y) = \alpha \|x - y\|^2$), then the "many small moves" option becomes the unique minimizer.

Interestingly, while mathematicians prefer to work with convex cost functions, economists prefer concave ones. The intuitive justification for this is that once goods have been loaded on to, say, a goods train, transporting the goods 200 kilometres costs much less than twice what it would cost to transport them 100 kilometres. Concave cost functions represent this economy of scale.

Abstract formulation of the problem

Monge and Kantorovich formulations

The transportation problem as it is stated in modern or more technical literature looks somewhat different because of the development of Riemannian geometry and measure theory. The mines-factories example, simple as it is, is a useful reference point when thinking of the abstract case. In this setting, we allow the possibility that we may not wish to keep all mines and factories open for business, and allow mines to supply more than one factory, and factories to accept iron from more than one mine.

Let X and Y be two separable metric spaces such that any probability measure on X (or Y) is a Radon measure (i.e. they are Radon spaces). Let $c : X \times Y \to [0, +\infty]$ be a Borel-measurable function. Given probability measures μ on X and ν on Y, Monge's formulation of the optimal transportation problem is to find a transport map $T : X \to Y$ that realizes the infimum

$$\inf \left\{ \int_X c(x, T(x))\, \mathrm{d}\mu(x) \,\middle|\, T_*(\mu) = \nu \right\},$$

where $T_*(\mu)$ denotes the push forward of μ by T. A map T that attains this infimum (i.e. makes it a minimum instead of an infimum) is called an "optimal transport map".

Monge's formulation of the optimal transportation problem can be ill-posed, because sometimes there is no T satisfying $T_*(\mu) = \nu$: this happens, for example, when μ is a Dirac measure but ν is not).

We can improve on this by adopting Kantorovich's formulation of the optimal transportation problem, which is to find a probability measure γ on $X \times Y$ that attains the infimum

$$\inf \left\{ \int_{X \times Y} c(x, y)\, \mathrm{d}\gamma(x, y) \,\middle|\, \gamma \in \Gamma(\mu, \nu) \right\},$$

where $\Gamma(\mu, \nu)$ denotes the collection of all probability measures on $X \times Y$ with marginals μ on X and ν on Y. It can be shown[3] that a minimizer for this problem always exists when the cost function X is lower semi-continuous and $\Gamma(\mu, \nu)$ is a tight collection of measures (which is guaranteed for Radon spaces X and Y). (Compare this formulation with the definition of the Wasserstein metric W_1 on the space of probability measures.)

Duality formula

The minimum of the Kantorovich problem is equal to

$$\sup \left(\int_X \phi(x)\, \mathrm{d}\mu(x) + \int_Y \psi(y)\, \mathrm{d}\nu(y) \right),$$

where the supremum runs over all pairs of bounded and continuous functions $\phi : X \to \mathbb{R}$ and $\psi : Y \to \mathbb{R}$ such that

$$\phi(x) + \psi(y) \leq c(x, y).$$

Solution of the problem

Optimal transportation on the real line

For $1 \leq p < +\infty$, let $\mathcal{P}_p(\mathbb{R})$ denote the collection of probability measures on $\mathbb{R}$ that have finite pth moment. Let $\mu, \nu \in \mathcal{P}_p(\mathbb{R})$ and let $c(x, y) = h(x - y)$, where $h : \mathbb{R} \to [0, +\infty)$ is a convex function.

1. If μ has no atom, i.e., if the cumulative distribution function $F_\mu : \mathbb{R} \to [0, 1]$ of μ is a continuous function, then $F_\nu^{-1} \circ F_\mu : \mathbb{R} \to \mathbb{R}$ is an optimal transport map. It is the unique optimal transport map if h is strictly convex.
2. We have

$$\min_{\gamma \in \Gamma(\mu,\nu)} \int_{\mathbb{R}^2} c(x, y)\, \mathrm{d}\gamma(x, y) = \int_0^1 c\left(F_\mu^{-1}(s), F_\nu^{-1}(s)\right)\, \mathrm{d}s.$$

Separable Hilbert spaces

Let X be a separable Hilbert space. Let $\mathcal{P}_p(X)$ denote the collection of probability measures on X such that have finite pth moment; let $\mathcal{P}_p^r(X)$ denote those elements $\mu \in \mathcal{P}_p(X)$ that are **Gaussian regular**: if g is any strictly positive Gaussian measure on X and $g(N) = 0$, then $\mu(N) = 0$ also.
Let $\mu \in \mathcal{P}_p^r(x),\ \nu \in \mathcal{P}_p(X),\ c(x, y) = |x - y|^p/p$ for $p \in (1, +\infty),\ p^{-1} + q^{-1} = 1$. Then the Kantorovich problem has a unique solution κ, and this solution is induced by an optimal transport map: i.e., there exists a Borel map $r \in L^p(X, \mu; X)$ such that

$$\kappa = (\mathrm{id}_X \times r)_*(\mu) \in \Gamma(\mu, \nu).$$

Moreover, if ν has bounded support, then

$$r(x) = x - x|\nabla\phi(x)|^{q-2}\nabla\phi(x) \text{ for } \mu\text{-almost all } x \in X$$

for some locally Lipschitz, *c*-concave and maximal Kantorovich potential ϕ. (Here $\nabla\phi$ denotes the Gâteaux derivative of ϕ.)

See also

- Wasserstein metric

References

[1] G. Monge. *Mémoire sue la théorie des déblais et de remblais. Histoire de l'Académie Royale des Sciences de Paris, avec les Mémoires de Mathématique et de Physique pour la même année*, pages 666–704, 1781.

[2] L. Kantorovich. *On the translocation of masses.* C.R. (Doklady) Acad. Sci. URSS (N.S.), 37:199–201, 1942.

[3] L. Ambrosio, N. Gigli & G. Savaré. *Gradient Flows in Metric Spaces and in the Space of Probability Measures.* Lectures in Mathematics ETH Zürich, Birkhäuser Verlag, Basel. (2005)

Article Sources and Contributors

Heuristic *Source*: http://en.wikipedia.org/w/index.php?oldid=359837884 *Contributors*: 2D, Aapo Laitinen, Ahoerstemeier, Alexbrewer, Alice, Altenmann, Alusayman, Ancheta Wis, AndriuZ, Andy120290, Anomalocaris, Anthony, Bcasterline, Belasted, BenKovitz, Bennó, Beve, Bladestorm, Bobblewik, Bookuser, Charles Matthews, CharlesGillingham, Chendols, Clossius, Cogpsych, Colfer2, Conejorojo, Cyan, Cyrius, DCDuring, Danno uk, Debeo Morium, Deepstratagem, ELApro, El C, Elizabeth84, Email4mobile, Epashou, Fixmacs, Fordmadoxfraud, Fplay, Fred Bauder, Fuzzform, Giftlite, Gilgamesh, Gogo Dodo, Grantsky, Grivantian, HappyDog, It Is Me Here, J. Finkelstein, J.delanoy, Jclemens, JimmyShelter, Joconnor, Johnkarp, Jonathan.s.kt, Jose Ramos, Joshhoyt97, Joychowdhury2009, K.C. Tang, K.anvesh87, KnightRider, Kwamikagami, LX, Leafpeeper, Lee Daniel Crocker, LilHelpa, Loren.wilton, Lothar76, LoveMonkey, Lucidish, Luna Santin, Lunaverse, MKil, Marcieri, Marco Krohn, Martarius, MartinPoulter, Matthew Stannard, Maximus Rex, Mellum, Michael Hardy, MrYdobon, Mssetiadi, Muspilli, Nanshu, Nbarth, Nectarflowed, Neutrality, Newbyguesses, Nikai, Noetica, Nomad, Oaktreedave, Orcrist, Paul Niquette, Penni17, Peterlewis, Philip Trueman, Philosotox, Pinkadelica, Pointblankstare, Qazmun, Quest for Truth, RadicalBender, Rankiri, Rdummarf, RedHouse18, RexNL, Robert K S, Rocko1124, Romanm, RoyalTS, Rrius, S Roper, Salgueiro, Samwaltz, Schmeitgeist, Shantavira, Simon Kilpin, SlackerMom, Snoyes, Spiral5800, Spitfire, Sslevine, Stephen378, Stereotek, Stevertigo, Sundar, TRMc, Taak, Tcncv, TeamZissou, Tenmei, The Anome, Theonemacduff, Thesilverbail, Thorwald, Thumperward, Tide rolls, Timneu22, Tnolley, Tompsci, Troped, Typofier, UkPaolo, Uncle Dick, Updatehelper, Varlaam, Victor Chmara, VinceyB, Waveguy, Weerasad, Wereon, Whoosit, Wik, Wolfdog, XJamRastafire, Xaxafrad, Xosé, Xp54321, Yakovsh, 266 anonymous edits

Combinatorial optimization *Source*: http://en.wikipedia.org/w/index.php?oldid=353873764 *Contributors*: Aliekens, Altenmann, Arnab das, Ben pcc, Ben1220, Bonniesteiglitz, Brunato, Brunner ru, CharlesGillingham, Cngoulimis, Cobi, Ctbolt, Daveagp, Diomidis Spinellis, Docu, Estr4ng3d, Giftlite, Giraffedata, Hike395, Jcc1, Kinema, Ksyrie, Lepikhin, Mellum, Michael Hardy, Miym, Moxon, Nocklas, Pjrm, RKUrsem, Remuel, RobinK, Sdorrance, Silverfish, StoneIsle, ThomHImself, Tizio, Tomo, Tribaal, Unara, 25 anonymous edits

Slack variable *Source*: http://en.wikipedia.org/w/index.php?oldid=346261906 *Contributors*: Autarch, HenningThielemann, Lourakis, Nbarth, Saravask, Xetxo, 1 anonymous edits

Constraint (information theory) *Source*: http://en.wikipedia.org/w/index.php?oldid=278092674 *Contributors*: Asymmetric, CzarNick, Dfass, Elonka, Gurch, KathrynLybarger, LAX, NHRHS2010, 7 anonymous edits

Problem *Source*: http://en.wikipedia.org/w/index.php?oldid=359938791 *Contributors*: 7girl7-aloha, Alansohn, Aldaron, Ancheta Wis, Andrejj, AndrewHowse, AndriuZ, Arichnad, Blueglow, Bonadea, Borgx, Bryan Derksen, Calvin 1998, Can't sleep, clown will eat me, Canon, Colourneon, ContactKiTFoX, D. Recorder, Dgaubin, DoubleBlue, Dreadstar, Edgar181, Ejosse1, Fbifriday, Fieldday-sunday, Fraggle81, Fredrick day, Gaidheal1, Gavia immer, Glenn, Gubbubu, Hraefen, Hyacinth, IOH, Imperial Monarch, Interiot, Inwind, Ixfd64, JPINFV, Jmf146, Joeahoff, Jose Icaza, Jpbowen, Levineps, Lukenigs, MONGO, Malcolma, Marc Venot, Materialscientist, Maxí, Mcapo13, Metadat, Mohsin-m, NAHID, NJMaxton, NawlinWiki, Neo-Jay, Nick Connolly, Nullifier128, Pgan002, PhJ, Pinethicket, Quackdave, Revolver, Rigadoun, Sabbut, Saheballah, Salgueiro, Shiftchange, Sjö, Slaveleader43, Smile a While, SoCalSuperEagle, Spencerk, Ss5te5t, Stephenb, Stevertigo, Sunray, Template namespace initialisation script, Thechangerchanges, Thedan1000, Thewayforward, Thingg, Tizio, Tjfulopp, Tomo, Vuara, Warrior50879, Wereon, Wolfdog, Woohookitty, Wrathchild, Wysprgr2005, Zzuuzz, 139 anonymous edits

Problem solving *Source*: http://en.wikipedia.org/w/index.php?oldid=358238105 *Contributors*: 2D, 2spyra, AbsolutDan, Action potential, Adapt, AdjustShift, Agbormbai, Ahoerstemeier, Alex Kosorukoff, Ancheta Wis, AndriuZ, Angr, Anna512, Antaeus Feldspar, Ap, Arjun G. Menon, Astral, Ayda D, Bachcell, Beanary, Billfmsd, Bobblewik, Boxplot, BrotherGeorge, Bufar, Celestianpower, ChemGardener, Cielomobile, Clhtnk, Cmcurrie, Coachuk, Corza, Credible58, D. Tchurovsky, DBragagnolo, Dave6, Dawn Bard, Dbfirs, Ddr, Derekristow, Djcremer, Djdrew76, DoctorW, Download, Eric-Wester, Eve Teschlemacher, Everyking, Extremecircuitz, Facilitation Author, Ferris37, Forenti, Frazzydee, Fvw, Genegorp, Giftlite, Gioto, GlassCobra, Good Vibrations, Graham87, Guggenheim, Gunnar Hendrich, HGB, Harald.schaub, Hari, Heron, Houghton, Hu12, Hydrogen Iodide, IceCreamAntisocial, Iggynelix, Ingi.b, Inter, IvR, IvanLanin, JForget, JaGa, Jacopo, Jan Tik, Jaranda, Jd2718, Jeames, JimmyShelter, JoachimFunke, JoeC 4321, Johnkarp, Jon Awbrey, Jose Icaza, Jrmindlesscom, Jtneill, Juanscott, Kazooou, Keilana, KillerChihuahua, Kizor, Kku, Kosebamse, Ksyrie, LaurensvanLieshout, Laxerf, Leestubbs, Letranova, Leuko, Levineps, Liao, LilHelpa, Lord Bane, Lupin, MCrawford, MMSequeira, Majorclanger, Marc Venot, Marek69, Markstrom, Marsbound2024, Marx01, Mattisse, Matty Austwick, Mbonline, McGeddon, Merlion444, Mgret07, Michael Hardy, Michaeldeutch, Mild Bill Hiccup, Moeron, Mr.Z-man, Nesbit, New Thought, Nicearma, Nikolas Stephan, Novum, Oda Mari, Oleg Alexandrov, Ott, Paranomia, ParisianBlade, Pavel Vozenilek, Pedant17, Peter gk, Peterlewis, Philip Trueman, Pickledh, Pilgrim27, Pioneer-12, Possum, Ralphruyters, Rapdetty, RaseaC, Reinyday, Richard001, RichardF, Rick Sidwell, Rjwilmsi, Ronz, Rsrikanth05, Ruud Koot, SLcomp, Saga City, Samhere, Saxifrage, Scoutersig, Sengkang, Sg gower, Shanes, SheldonN, Shimgray, Shoessss, Sidaki, Spalding, Srikeit, Stifle, Stone, Suidafrikaan, Tagishsimon, Tarquin, TexasAndroid, The Evil IP address, TheEgyptian, Theresa knott, Thiseye, ThreePD, Vaughan, VodkaJazz, VoteFair, Whatfg, Wikilibrarian, Wimbojambo, Woohookitty, Yammy Yamathorn, YellowMonkey, Yhkhoo, ZimZalaBim, Ziphon, 252 anonymous edits

Linear search problem *Source*: http://en.wikipedia.org/w/index.php?oldid=350465584 *Contributors*: Decltype, RobinK, Shuroo, TrulyBlue, 2 anonymous edits

Linear complementarity problem *Source*: http://en.wikipedia.org/w/index.php?oldid=353745196 *Contributors*: ARdolf, Ashwin, Charles Matthews, Delaszk, Henrygb, Iorsh, JYOuyang, Kiefer.Wolfowitz, Michael Hardy, Nagle, Numsgil, Pak21, Silverfish, Thrufir, TotientDragooned, Trevorgoodchild, Unara, 9 anonymous edits

Mixed linear complementarity problem *Source*: http://en.wikipedia.org/w/index.php?oldid=302452612 *Contributors*: Numsgil

Boolean satisfiability problem *Source*: http://en.wikipedia.org/w/index.php?oldid=357084624 *Contributors*: 151.99.218.xxx, Ahalwright, Alex R S, AndrewHowse, Ap, Artem M. Pelenitsyn, B4hand, Brion VIBBER, CBM, CRGreathouse, Chalst, ChangChienFu, Conversion script, Creidieki, DBeyer, DFRussia, Damian Yerrick, David.Monniaux, Dcoetzee, Doomdayx, Drbreznjev, Dwheeler, Dysprosia, Elias, Enmiles, Everyking, Fancieryu, FlashSheridan, Fratrep, Gdr, Giftlite, Gregbard, Guarani.py, Gwern, Hans Adler, Hattes, Hh, Igor Markov, J. Finkelstein, Jan Hidders, Janto, Jecar, Jimbreed, Jok2000, Jon Awbrey, Jpbowen, Julian Mendez, Karada, Karl-Henner, LC, Leibniz, Localzuk, LouScheffer, Luuva, Maghnus, Mamling, MathMartin, Max613, McCart42, Mellum, Mets501, Mhym, Michael Hardy, Michael Shields, Miym, Mjuarez, Mmn100, Mqasem, Mutilin, Naddy, Night Gyr, Nilmerg, Obradovic Goran, Oerjan, Oliver Kullmann, PoeticVerse, PsyberS, Quaternion, RG2, Saforrest, Sam Hocevar, Schneelocke, Siteswapper, Sl, Tim Starling, Timwi, Tizio, Twiki-walsh, Vegasprof, Weichaoliu, Wik, Wvbailey, Yury.chebiryak, Yworo, Zander, Zarrapastro, Zeno Gantner, ZeroOne, Станиславъ, 118 anonymous edits

Mathematical problem *Source*: http://en.wikipedia.org/w/index.php?oldid=326446516 *Contributors*: BL, Booyabazooka, Bryan Derksen, Cyan, Doulos Christos, Fox, Giftlite, Greenrd, H.ehsaan, Hao2lian, Haonhien, Karl-Henner, Karlscherer3, Lambiam, Linas, MCrawford, Malcolm Farmer, Martian.knight, Mathematrucker, Mdd, Meelar, Mhaitham.shammaa, Michael Hardy, Nic bor, Niteowlneils, Ohnoitsjamie, Oleg Alexandrov, PV=nRT, Pgan002, Pizza Puzzle, Rettetast, Rybu, Salgueiro, Shiftchange, Theresa knott, Thomasmeeks, Timwi, Tom Lougheed, Trick, Uarrin, ZeroOne, Zwittermaedchen, 20 anonymous edits

Travelling salesman problem *Source*: http://en.wikipedia.org/w/index.php?oldid=359861997 *Contributors*: 130.233.251.xxx, 62.202.117.xxx, ANONYMOUS COWARD0xC0DE, Aaronbrick, Adammathias, Ahoerstemeier, Akokskis, Alan.ca, Aldie, Altenmann, Andreas Kaufmann, Andris, Angus Lepper, Apanag, ArglebargleIV, Astral, AstroNomer, B4hand, Bathysphere, BenjaminTsai, Bensin, Bjornson81, Bo Jacoby, Bongwarrior, Boothinator, Brian Gunderson, Brucevdk, Brw12, C. lorenz, CRGreathouse, Can't sleep, clown will eat me, Capricorn42, ChangChienFu, Chris-gore, ChrisCork, Classicalecon, Cngoulimis, Coconut7594, Conversion script, CountingPine, Da monster under your bed, Daniel Karapetyan, David Eppstein, David.Mestel, David.hillshafer, DavidBiesack, Dbfirs, Dcoetzee, Devis, Dino, Disavian, Donarreiskoffer, Doradus, Downtown dan seattle, DragonflySixtyseven, DreamGuy, Dwhdwh, Dysprosia, Edward, El C, Ellywa, Fanis84, Ferris37, Fmccown, Fredrik, French Tourist, Gaeddal, Galoubet, Gdr, Giftlite, Gnomz007, Gogo Dodo, Greenmatter, H, Hairy Dude, Hans Adler, Haterade111, Hawk777, Herbee, Hike395, Honnza, Hyperneural, Irrevenant, Isaac, IstvanWolf, IvR, Ixfd64, J.delanoy, JackH, Jackbars, Jamesd9007, Jasonb05, Jeffhoy, Jim.Callahan,Orlando, Johngouf85, Jok2000, JonathanFreed, Jsamarziya, Jugander, KGV, Kane5187, Karada, Kenneth M Burke, Kenyon, Kf4bdy, Kjells, Klausikm, Kotasik, Kri, Ksana, Kvamsi82, Kyokpae, LFaraone, Lambiam, Laudaka, Lingwanjae, MSGJ, MagicMatt1021, Male1979, Mantipula, MarSch, Marj Tiefert, Martynas Patasius, Materialscientist, MathMartin, Mdd, Mellum, Melsaran, Mhahsler, Michael Hardy, Michael Slone, Miym, Monstergurkan, MoraSique, Mormegil, Musiphil, Mzamora2, Nethgirb, Nguyen Thanh Quang, Ninjagecko, Nobbie, Nr9, Obradovic Goran, Orfest, Paul Silverman, Pauli133, PeterC, Petrus, Pgr94, Phcho8, Piano non troppo, PierreSelim, Pleasantville, Pmdboi, Qaramazov, Qorilla, Quadell, R3m0t, Random contributor, Ratfox, Raul654, RedLyons, Requestion, Rheun, Richmeister, Rjwilmsi, RobinK, Rocarvaj, Ronaldo, Rror, Ruakh, Ruud Koot, Ryan Roos, STGM, Saeed.Veradi, Sahuagin, Seet82, Seraphimblade, Shadowjams, Sharcho, ShelfSkewed, Shoujun, Siddhant, Simetrical, Sladen, Smmurphy, Smremde, Smyth, Soupz, SpNeo, SpuriousQ, Stemonitis, Stimpy, Stochastix, StradivariusTV, Superm401, Superninja, Tamfang, Teamtheo, Tedder, That Guy, From That Show!, The Anome, The Thing That Should Not Be, The stuart, Theodore Kloba, Thisisbossi, Thore Husfeldt, Tigerqin, Tinman, Tomgally, Tsplog, Twas Now, Vasil', Vgy7ujm, WhatisFeelings?, Wizard191, Wumpus3000, Wwwwolf, Xiaojeng, Yixin.cao, Zaphraud, Zeno Gantner, ZeroOne, Zyqqh, 452 anonymous edits

Knapsack problem *Source*: http://en.wikipedia.org/w/index.php?oldid=359956266 *Contributors*: -- April, Acipsen, Acroterion, Altenmann, Andycjp, Angela, Aquatopia, Arcturus4669, Austinjp, Bluebusy, Carbuncle, Catslash, Cngoulimis, Conversion script, Cronholm144, Daggerbox, Dantheox, Datakid, Daveagp, David Eppstein, DavidCary, Dcoetzee, Edmundgreen, Faridani, Feofiloff, FrankTobia, Gandalf61, Geoffrey, Giftlite, Gilliam, Honza Záruba, Hégésippe Cormier, Ixfd64, Jitse Niesen, Knapsack problem, Kubieziel, LC, Larry_Sanger, Leeannedy, Leonard G., LiranKatzir, Longhair, Madmik3, MathMartin, Matt Crypto, Mdd, Mellum, Michael Hardy, Michael miceli, Offliner, Offwiki, Phil Boswell, Px21, Qwertyus, Rbirmann, Rfl, Robertvan1, Rohitm 001, Rspeer, RxS, Sahuagin, Starwiz, Stephen Gilbert, Stevertigo, Taejo, Tarotcards, TheEternalVortex, Thijswijs, Thr4wn, Timwi, Tobias Bergemann, Triponi, Twri, Unara, Unix68, Wikipelli, Witger, XDanielx, 126 anonymous edits

Nonlinear programming *Source*: http://en.wikipedia.org/w/index.php?oldid=359044685 *Contributors*: Ajgorhoe, Alexander.mitsos, BarryList, Broom eater, Brunner7, Charles Matthews, Dmitrey, Dto, EconoPhysicist, EdJohnston, EncMstr, Frau Holle, FrenchIsAwesome, G.de.Lange, Giftlite, Hike395, Hu12, Jamelan, Jaredwf, Jitse Niesen, Kiefer.Wolfowitz, KrakatoaKatie,

Leonard G., McSush, Mcmlxxxi, Mdd, Metiscus, Miaow Miaow, Monkeyman, MrOllie, Myleslong, Nacopt, Oleg Alexandrov, Olegalexandrov, PimBeers, Psvarbanov, RekishiEJ, Sabamo, Stevenj, Tgdwyer, User A1, 45 anonymous edits

Assignment problem *Source*: http://en.wikipedia.org/w/index.php?oldid=330542896 *Contributors*: AZhnaZg, Altenmann, Andrewa, Anonymoues, Charles Matthews, David Eppstein, Evercat, Fnielsen, Infrogmation, Jklin, Karipuf, Klahnako, Lage, Lambiam, Lawsonsj, Michael Hardy, Michael Slone, MichaelGensheimer, Miym, Nils Grimsmo, Oleg Alexandrov, Onebyone, PAS, Paul Stansifer, Pearle, RJFJR, Rogerb67, Sciurinæ, SimonP, Teknic, The Anome, Tribaal, Vikrams jamnal, Wikid77, Wshun, 44 anonymous edits

Decision problem *Source*: http://en.wikipedia.org/w/index.php?oldid=350829942 *Contributors*: 128.138.87.xxx, Arthena, Ascánder, AxelBoldt, Baiji, CBM, CRGreathouse, Chinju, Conversion script, Creidieki, Culix, Cwitty, David.Monniaux, Dcoetzee, Dlakavi, Drae, Dratman, Ehn, Eiji adachi, Gdr, Giftlite, Giraffedata, Gregbard, Hadal, Inkling, Iseeaboar, Jafet, Jonathan de Boyne Pollard, Kesla, Kragen, Krymson, Kurykh, Kuszi, LC, Lalaith, Lambiam, MIT Trekkie, Mandarax, MathMartin, Mellum, MementoVivere, NekoDaemon, Noroton, Nortexoid, Obradovic Goran, Od Mishehu, Oleg Alexandrov, Pakaran, Paul August, Philgp, Pichpich, Pohta ce-am pohtit, Pro8, RobinK, Salsa Shark, SalvNaut, Seb, Sho Uemura, Shreevatsa, Sligocki, Stevertigo, UberScienceNerd, Uday, Unixxx, UsaSatsui, Wavelength, Wvbailey, Ylloh, 36 anonymous edits

Proof theory *Source*: http://en.wikipedia.org/w/index.php?oldid=358990776 *Contributors*: Arthur Rubin, Brian0918, Bryan Derksen, CBM, CBM2, Cabe6403, Chalst, Charles Matthews, Comiscuous, David Eppstein, Dbtfz, Dicklyon, Dima125, Dominus, Dysprosia, Gene Nygaard, Giftlite, Gregbard, Hairy Dude, JRB-Europe, JRSpriggs, Jahiegel, Jni, Jorend, Jtauber, Kntg, Krappie, LaForge, Lambiam, Leibniz, Llywrch, Luqui, Magmi, Mannypabla, Markus Krötzsch, MattTait, Mav, Michael Hardy, Msh210, Nortexoid, Number 0, Pj.de.bruin, Porcher, Qxz, Rizome, Rotem Dan, Tbvdm, The Anome, Thisthat12345, Tillmo, Toby Bartels, Tong, Yonaa, Youandme, 47 anonymous edits

Optimization problem *Source*: http://en.wikipedia.org/w/index.php?oldid=349941423 *Contributors*: Altenmann, C. lorenz, Charles Matthews, Culix, Giftlite, Hermel, Mellum, Michael Hardy, Nono64, Obradovic Goran, Pohta ce-am pohtit, Rich Farmbrough, Rinconsoleao, Rjwilmsi, RobinK, The Anome, Wavelength, 5 anonymous edits

Constraint satisfaction problem *Source*: http://en.wikipedia.org/w/index.php?oldid=346303365 *Contributors*: Alai, AndrewHowse, BACbKA, Beland, Coneslayer, David Eppstein, Delirium, Dgessner, Diego Moya, DracoBlue, Ertuocel, Jamelan, Jdpipe, Jgoldnight, Jkl, Jradix, Karada, Mairi, MarSch, Michael Hardy, Ogai, Oleg Alexandrov, Oliphaunt, Ott2, Patrick, Rl, Simeon, The Anome, Tizio, Uncle G, 22 anonymous edits

Algorithm *Source*: http://en.wikipedia.org/w/index.php?oldid=359767881 *Contributors*: "alyosha", 0, 12.35.86.xxx, 128.214.48.xxx, 151.26.10.xxx, 161.55.112.xxx, 204.248.56.xxx, 24.205.7.xxx, 747fzx, 84user, 98dblachr, APH, Aarandir, Abovechief, Abrech, Acroterion, Adam Marx Squared, Adamarthurryan, Adambiswanger1, Addshore, Aekamir, Agasta, Agent phoenex, Ahy1, Alcalazar, Ale2006, Alemua, Alexandre Bouthors, Alexius08, Algogeek, Allan McInnes, Amberdhn, AndonicO, Andre Engels, Andreas Kaufmann, Andrejj, Andres, Anomalocaris, Antandrus, Anthony, Anthony Appleyard, Anwar saadat, Apofisu, Arvindn, Athaenara, AxelBoldt, Azurgi, B4hand, Bact, Bart133, Bb vb, Benn Adam, Bethnim, Bill4341, BillC, Billcarr178, Blankfaze, Bob1312, Bobblewik, Boing! said Zebedee, BorgQueen, Boud, Brenont, BriEnBest, Brion VIBBER, Brutannica, Bryan Derksen, Bth, Bucephalus, CBM, CRGreathouse, Cameltrader, CarlosMenendez, Cascade07, Cbdorsett, Cedars, Chadernook, Chamal N, Charles Matthews, Charvex, Chasingsol, Chatfecter, Chinju, Chris 73, Chris Roy, Citneman, Ckatz, Clarince63, Closedmouth, Cmdieck, Colonel Warden, Conversion script, Cornflake pirate, Corti, CountingPine, Crazysane, Cremepuff222, Curps, Cybercobra, Cyberjoac, DASSAF, DCDuring, Danakil, Daven200520, David Eppstein, David Gerard, Dbabbitt, Dcoetzee, DeadEyeArrow, Deadcracker, Deeptrivia, Delta Tango, Den fjättrade ankan, Deor, Depakote, DerHexer, Derek farn, DevastatorIIC, Dgrant, Dinsha 89, Discospinster, DopefishJustin, Dreftymac, Drilnoth, DslIWG,UF, Duncharris, Dwheeler, Dylan Lake, Dysprosia, EconoPhysicist, Ed Poor, Ed g2s, Editorinchief1234, Eequor, Efflux, El C, ElectricRay, Electron9, ElfMage, Ellegantfish, Eloquence, Emadfarrokhih, Epbr123, Eric Wester, Eric.ito, Erik9, Essjay, Eubulides, Everything counts, Evil saltine, EyeSerene, Fabullus, Fantom, Farosdaughter, Farshadrbn, Fastfission, Fastilysock, Fernkes, FiP, FlyHigh, Fragglet, Frecklefoot, Fredrik, Friginator, Frikle, GOV, GRAHAMUK, Gaius Cornelius, Galoubet, Gandalf61, Geniac, Geo g guy, Geometry guy, Ghimboueils, Gianfranco, Giantscoach55, Giftlite, Gilgamesh, Giminy, Gimme danger, Gioto, Gogo Dodo, Goochelaar, Goodnightmush, Googl, GraemeL, Graham87, Gregbard, Groupthink, Grubber, Gubbubu, Gurch, Guruduttmallapur, Guy Peters, Guywhite, H3l1x, Hadal, Hairy Dude, Hamid88, Hannes Eder, Hannes Hirzel, Harryboyles, Harvester, HenryLi, HereToHelp, Heron, Hexii, Hfastedge, Hiraku.n, Hu12, Hurmata, Iames, Ian Pitchford, Imfa11ingup, Inkling, InterruptorJones, Intgr, Iridescent, Isis, Isofox, Ixfd64, J.delanoy, JForget, JIP, JSimmonz, Jacomo, Jacoplane, Jagged 85, Jaredwf, Jeff Edmonds, Jeronimo, Jersey Devil, Jerzy, Jidan, JoanneB, Johan1298, Johantheghost, Johneasley, Johnsap, Jojit fb, Jonik, Jonpro, Jorvik, Josh Triplett, Jpbowen, Jtvisona, Jusdafax, Jóna Þórunn, K3fka, KHamsun, Kanags, Kanjy, Kanzure, Keilana, Kenbei, Kevin Baas, Kh0061, Khakbaz, Kku, Kl4m, Klausness, Kntg, Kozuch, Kragen, Krellis, LC, Lambiam, LancerSix, Larry R. Holmgren, Ldo, Ldonna, Levineps, Lexor, Lhademmor, Lightmouse, Lilwik, Ling.Nut, Lissajous, Lumidek, Lumos3, Lupin, Luís Felipe Braga, MARVEL, MSPbitmesra, MagnaMopus, Makewater, Makewrite, Maldoddi, Malleus Fatuorum, Mange01, Mani1, ManiF, Manik762007, Marek69, Mark Dingemanse, Markaci, Markh56, Markluffel, Marysunshine, MathMartin, Mathviolintennis, Matt Crypto, MattOates, Mav, Maxamegalon2000, McDutchie, Meowist, Mfc, Michael Hardy, Michael Slone, Michael Snow, MickWest, Miguel, Mikeblas, Mindmatrix, Mission2ridews, Miym, Mlpkr, Mpeisenbr, MrOllie, Mttcmbs, Multipundit, MusicNewz, MustangFan, Mxn, Nanshu, Napmor, Nikai, Nikola Smolenski, Nil Einne, Nmnogueira, Noisy, Nummer29, Obradovic Goran, Od Mishehu, Odin of Trondheim, Ohnoitsjamie, Onorem, OrgasGirl, Orion11M87, Ortolan88, Oskar Sigvardsson, Oxinabox, Oxymoron83, P Carn, PAK Man, PMDrive1061, Paddu, PaePae, Pascal.Tesson, Paul August, Paul Foxworthy, Paxinum, Pb30, Pde, Penumbra2000, Persian Poet Gal, Pgr94, Philip Trueman, Pit, Plowboylifestyle, Pohta ce-am pohtit, Poor Yorick, Populus, Possum, PradeepArya1109, Quendus, Quintote, Quota, Qwertyus, R. S. Shaw, Raayen, RainbowOfLight, Randomblue, Raul654, Rdsmith4, Reconsider the static, Rejka, Rettetast, RexNL, Rgoodermote, Rholton, Riana, Rich Farmbrough, Rizzardi, RobertG, RobinK, Rpwikiman, Rror, RussBlau, Ruud Koot, Ryguasu, SJP, Salix alba, Salleman, SamShearman, SarekOfVulcan, Savidan, Scarian, Seb, Sesse, Shadowjams, Shipmaster, Silly rabbit, SilverStar, Sitharama.iyengar1, SlackerMom, Snowolf, Snoyes, Sonjaaa, Sophus Bie, Sopoforic, Spankman, Speck-Made, Spellcast, Spiff, Splang, Sridharinfinity, Stephan Leclercq, Storkk, Sundar, Susurrus, Swerdnaneb, Systemetsys, TakuyaMurata, Tarquin, Taw, The Firewall, The Fish, The Thing That Should Not Be, The ansible, TheGWO, TheNewPhobia, Thecarbanwheel, Theodore7, Tiddly Tom, Tide rolls, Tim Marklew, Timc, Timhowardriley, Timir2, Tizio, Tlesher, Tlork Thunderhead, Toncek, Tony1, Trevor MacInnis, Treyt021, TuukkaH, UberScienceNerd, Urenio, User A1, V31a, Vasileios Zografos, Vikreykja, Vildricianus, Wainkelly, Waltnmi, Wavelength, Wayiran, Waynefan23, Weetoddid, Wellithy, Wexcan, Who, Whosyourjudas, WhyDoIKeepForgetting, WikHead, Willking1979, Winston365, Woohookitty, Wvbailey, Xashaiar, Yamamoto Ichiro, Yintan, Ysindbad, Zfr, Zocky, Zondor, Zoney, Zundark, 850 anonymous edits

Approximation algorithm *Source*: http://en.wikipedia.org/w/index.php?oldid=351770055 *Contributors*: Andris, Arthena, Ashishgoel.1973, Bmatheny, C. lorenz, Culix, DKalkin, Danny, David Eppstein, Dcoetzee, Decrease789, Dricherby, Fredrik, Giftlite, Haham hanuka, Jnestorius, Kolyma, LachlanA, Mellum, NathanHurst, NotQuiteEXPComplete, Ojigiri, Oleg Alexandrov, Pnamjoshi, RMFan1, Ratfox, Rjwilmsi, RobinK, Ruud Koot, Tiagofassoni, Tribaal, Vavi, Whodoesthis, 32 anonymous edits

Search algorithm *Source*: http://en.wikipedia.org/w/index.php?oldid=359901607 *Contributors*: Alex.g, Alexius08, Alexmorter, Altenmann, Andromeda, AngleWyrm, AnotherPerson 2, Barneca, Baruneju, Bbik, Beland, Bogdangiusca, Boleslav Bobcik, Bonadea, Bsilverthorn, CYD, Capricorn42, CharlesGillingham, Chris G, ClansOfIntrigue, Colin Barrett, Conversion script, Cspooner, Dariopy, Daveagp, Dcoetzee, Decrypt3, DevastatorIIC, Diego Moya, Doc glasgow, EatMyShortz, Flandidlydanders, Fmicaeli, Frikle, GPHemsley, Gattom, Gene.arboit, Giftlite, Gregman2, HamburgerRadio, Jitse Niesen, Jorge Stolfi, Kc03, Kdakin, Kenyon, Kgeza7, Kragen, Lamdk, Ligulem, Mezzanine, Mikeblas, Mild Bill Hiccup, Mironearth, Mlpkr, Mundhenk, Nanshu, Nixdorf, Nohat, Ohnoitsjamie, Pgr94, Phils, Plasticup, Quadrescence, RW Marloe, Rgoodermote, Richardj311, RichiH, Robinh, SLi, Saaska, Sam Hocevar, Shuroo, SiobhanHansa, Snowolf, Spidern, Stefano, Taral, Taw, Tcncv, The12game, ThomHImself, Tide rolls, Tomaxer, Vassloff, Ww, Xijiahe, 133 anonymous edits

Greedy algorithm *Source*: http://en.wikipedia.org/w/index.php?oldid=359635048 *Contributors*: A8UDI, Andreas Kaufmann, ArzelaAscoli, AxelBoldt, Brianyoumans, CKlunck, CatherineMunro, Chamale, Charles Matthews, Cjohnzen, Clan-destine, CloudNine, Cruccone, David Eppstein, Dcoetzee, Diomidis Spinellis, Eirik the Viking, Eleschinski2000, Enmc, Enochlau, Giftlite, Haham hanuka, Hairy Dude, HairyFotr, Hammertime, Hfastedge, Hhulzk, Hobartimus, Jaredwf, Jddriessen, Jibbist, Kiefer.Wolfowitz, Kim Bruning, LOL, Malcohol, Mange01, Marcosw, Mcstrother, Mechonbarsa, Meduz, Meekywiki, Mindmatrix, Mlpkr, Nandhp, Nethgirb, New Thought, NickShaforostoff, Notheruser, Obradovic Goran, Pavel Kotrc, PeterBrooks, Pgan002, Que, Ralphy, Ruud Koot, Ryanmcdaniel, Salgueiro, Sango123, Smjg, Suanshsinghal, Sverdrup, Swapspace, SynergyBlades, TheMandarin, Trezatium, Unyoyega, Uselesswarrior, Wikid77, Wleizero, Wlievens, Zangkannt, ZeroOne, ایووپ, 83 anonymous edits

Divide and conquer algorithm *Source*: http://en.wikipedia.org/w/index.php?oldid=355879075 *Contributors*: Adam78, Ailun, Algoritmist, Amshali, AndrewKepert, Antzervos, ArbitUsername, Atomota, Bobo192, CL, Daniel Quinlan, David Eppstein, Dcoetzee, Destructor, DevastatorIIC, DragonFury, Dtrebbien, Ekaratsuba, El C, Excirial, Finem, Fredrik, Furrykef, Garion96, Giftlite, Goffrie, Head, IrishPete, Irrevenant, J.delanoy, Jake-helliwell, Jfcorbett, Jorge Stolfi, LanceBarber, LiDaobing, LilHelpa, Linas, Looxix, MIT Trekkie, Madmardigan53, Mdd, Mlpkr, Mycompsimm, Neg, Phe, Philip Trueman, Pichpich, Pomte, Poor Yorick, Quercus basaseachicensis, R. S. Shaw, Ruud Koot, Stevenj, Taichi, TakuyaMurata, Tardis, ThePedanticPrick, Thijswijs, Thingg, UTQ Shadow, Unyoyega, VKokielov, Varuna, WadeSimMiser, Wolfrock, XJamRastafire, 76 anonymous edits

Simplex algorithm *Source*: http://en.wikipedia.org/w/index.php?oldid=358414385 *Contributors*: Abhishek, Abovechief, Aegis Maelstrom, Afluent Rider, Akhram, Alexey.salnikov, Altenmann, Apavlo, Arthur Rubin, Barak, Baryonic Being, BenFrantzDale, Bob the third, Brian Parker, C S, CRGreathouse, Charles Matthews, Cjohnzen, Cnsdtzsf, CrazyTerabyte, David Eppstein, Dcoetzee, Denshade, Discospinster, Drini, Dungodung, Edurant, Enochlau, Epachamo, Gerdemb, Giftlite, Gsmgm, Guillorama, Hike395, Inframaut, J.delanoy, Jacj, Jajhall, Jaredwf, Jim.belk, Jitse Niesen, Jjbeard, JoanneB, JonMcLoone, KeithTyler, Kiefer.Wolfowitz, Lubos, Madmax.ptz, Mdd, Mghunt, Michael Hardy, Myleslong, NathanHurst, Nerd65536, New Thought, Nimur, Nmnogueira, Numsgil, Oleg Alexandrov, Oliphaunt, Orderud, Paul August, Pecondon, Peter Ballard, Petter Strandmark, Pfortuny, Pjvpjv, Predictor, Reject, Rgrimson, Ricardogobbo, Sanders muc, Sandskies, Shahab, Smartcat, Stebulus, Tbbooher, Tomerfiliba, Tong, TryamMan, Tutor dave, Twri, Vasil', VictorAnyakin, Viz, WAS 4.250, WebHamster, Wikid77, Xyz9000, Yoderj, 121 anonymous edits

Critical path method *Source*: http://en.wikipedia.org/w/index.php?oldid=355311402 *Contributors*: Achalmeena, Amarvk, Amgunite, Amientan, BBar, BlinderBomber, Blueboy96, CPMTutor, CallmeDrNo, Carolfrog, Chachrist, Cheese Sandwich, Cnbrb, Corpx, Darguz Parsilvan, Delaszk, Donreed, Dragmas, Elkinsra, Eubulides, Firien, G2opdtsl, George100, Graibeard, Harda, Hekerui, Hu12, Inwind, IrishInNY, Ivolution, Jamelan, Jbarmash, Jeltz, Kaihsu, Kenmckinley, Khoshino, Kuru, Kwhitten, Lindsay658, Lordkyl, MER-C, Mausy5043, Mdd, Michael Hardy, Mod.torrentrealm, NCurse, Nascar fan mx, NcSchu, Newman9997, Nixdorf, Nuggetkiwi, Oxymoron83, PatrickWeaver, PigFlu Oink, Pingveno, Pm master, Ppntori, Quuxplusone, RHaworth,

SNIyer12, Shlomke, Sin-man, SkipHuffman, Stevenwmccrary58, Thesydneyknowitall, Theuniversalcynic, Tijuana Brass, Vincehk, Vincnet, Vvkvaranasi, Woood, Wsmith202, 87 anonymous edits

Monte Carlo method *Source*: http://en.wikipedia.org/w/index.php?oldid=359697832 *Contributors*: *drew, A.Cython, ABCD, Aardvark92, Adfred123, Aferistas, Agilemolecule, Akanksh, Alanbly, Albmont, AlexBIOSS, AlexandreCam, AlfredR, Alliance09, Altenmann, Andrea Parri, Andreas Kaufmann, Angelbo, Aniu, Apanag, Aspuru, Atlant, Avalcarce, Avicennasis, Aznrocket, BAxelrod, BConleyEEPE, Banano03, Banus, Bduke, BenFrantzDale, BenTrotsky, Bender235, Bensaccount, BillGosset, Bkell, Blotwell, Bmaddy, Bobo192, Boffob, Boredzo, Broquaint, Btyner, CRGreathouse, Caiaffa, Charles Matthews, ChicagoActuary, Cibergili, Cm the p, Colonies Chris, Coneslayer, Cretog8, Criter, Cybercobra, Cython1, DMG413, Damistmu, Davnor, Ddcampayo, Ddxc, Digemedi, Drsquirlz, Ds53, Duck ears, Duncharris, Dylanwhs, ERosa, EldKatt, Elpincha, Elwikipedista, Eudaemonic3, Ezrakilty, Fastfission, Fintor, Flammifer, Frozen fish, Furrykef, G716, Giftlite, Gilliam, Goudzovski, GraemeL, GrayCalhoun, Greenyoda, Grestrepo, Gtrmp, Gökhan, Hanksname, Hawaiian717, Hokanomono, Hu12, Hubbardaie, ILikeThings, IanOsgood, Inrad, Itub, Jackal irl, Jacobleonardking, Janpedia, JavaManAz, Jeffq, Jitse Niesen, Joey0084, John, JohnOwens, Jorgenumata, Jsarratt, Jugander, Jérôme, K.lee, KSmrq, KaHa242, Karol Langner, Kenmckinley, Kimys, Knordlun, Kroese, Kummi, Kuru, Lambyte, LeoTrottier, Levin, Lexor, LizardJr8, LoveMonkey, M-le-mot-dit, Malatesta, Male1979, ManchotPi, Marcofalcioni, Mark Foskey, Martinp, Masatran, Mathcount, MaxHD, Maxentrope, Maylene, Mbryantuk, Melcombe, Michael Hardy, Mikael V, Misha Stepanov, Mlpearc, Mnath, Moink, Mtford, Nagasaka, Nanshu, Narayanese, Nasarouf, Nelson50, Nosophorus, Nsaa, Nuno Tavares, Nvartaniucsd, Ohnoitsjamie, Oli Filth, Oneboy, Orderud, OrgasGirl, Ott2, P99am, Paul August, PaulxSA, Pbroks13, Pcb21, Pete.Hurd, PeterBoun, Pgreenfinch, Philopp, Pibwl, Pinguin.tk, PlantTrees, Pne, Popsracer, Poupoune5, Qadro, Quantumelfmage, Quentar, Qxz, RWillwerth, Ramin Nakisa, Redgolpe, Renesis, Richie Rocks, Rinconsoleao, Rjmccall, Ronnotel, Rs2, SKelly1313, Sam Korn, Samratvishaljain, Sergio.ballestrero, Shacharg, Shreevatsa, Snegtrail, Snoyes, Somewherepurple, Spellmaster, Splash6, SpuriousQ, Stefanez, Stefanomione, StewartMH, Stimpy, Storm Rider, Superninja, Sweetestbilly, Tarantola, Taxman, Tayste, Tesi1700, Theron110, Thirteenity, ThomasNichols, Thr4wn, Tiger Khan, Tim Starling, Tom harrison, TomFitzhenry, Tooksteps, Trebor, Twooars, UBJ 43X, Urdutext, Uwmad, Vipuser, VoseSoftware, Wile E. Heresiarch, William Avery, Yoderj, Zarniwoot, Zoicon5, Zr40, Zuidervled, 359 anonymous edits

NP-complete *Source*: http://en.wikipedia.org/w/index.php?oldid=357182259 *Contributors*: 62.202.117.xxx, ANONYMOUS COWARD0xC0DE, Afrozenator, AmarChandra, Andris, AndyBQ, Anuragbms, Arthena, Arthur Frayn, Arvindn, As the glorious weep, Ascánder, AxelBoldt, B^4, Baderyp, Bartledan, Bewildebeast, BjarteSorensen, Booyabazooka, BozMo, Bubba73, Bwabes, CALR, CRGreathouse, CarlH, Chocolateboy, Clangin, Clecio, Coblin, Conversion script, Creidieki, Cwitty, Cyde, DMCer, Dark knight ita, Darth Mike, David Eppstein, David Haslam, Dcoetzee, Demonic1993, Deputyduck, Dominus, Dratman, Drostie, Duncancumming, Eewild, Eric119, Eubulides, Excirial, François Pitt, Fredanator, Gazpacho, Gdr, Giftlite, Gillyweed, Graham87, Imz, Indeterminate, Isomorph, Ixfd64, J04n, JRSpriggs, JamesGecko, Jan Hidders, Jimbreed, JoeKearney, Jok2000, Jon.c.anderson, Jschwa1, Kalathalan, Karl-Henner, Klausness, LC, Laurusnobilis, M412k, Maximus Rex, Mcsee, Mdkess, Mellum, Mishlai, Miym, Mmo, MoRsE, Mormegil, Nahum Reduta, Nick, Nitefood, Obradovic Goran, Octahedron80, Oleg Alexandrov, Olivier, Ott2, OwenX, Pakaran, Petri Krohn, Phlusko, Piet Delport, Pohta ce-am pohtit, Poor Yorick, Populus, Powo, Psiphiorg, Quuxplusone, Qwertyus, R3m0t, Retired username, Rhobite, Rjwilmsi, Rob Zako, Robert Merkel, RobinK, Rory096, Rotational, Seffer, Silvonen, Snoyes, Spock of Vulcan, Stevertigo, Sundar, Swordsmankirby, Taemyr, Takaitra, TakuyaMurata, Tarotcards, Template namespace initialisation script, ThG, That Guy, From That Show!, Thore, Timwi, Tizio, Tristanb, Trovatore, Ttiotsw, Tyler McHenry, Ultimus, Xcez-be, Ylloh, Zhefurui, 170 anonymous edits

Dynamic programming *Source*: http://en.wikipedia.org/w/index.php?oldid=359866414 *Contributors*: 1baumann, AHMartin, Abi79, Aceituno, Aeons, Alex.altmaster, Altenmann, Ancheta Wis, Apis O-tang, AshtonBenson, Atif.hussain, Babbage, Beefman, Beland, Bluebusy, Brentsmith101, Cancan101, Cannin, Chan siuman, Chipuni, Conskeptical, Crystallina, Cybercobra, D h benson, Damian.frank, David Eppstein, Dbroadwell, Dcoetzee, Drilnoth, Edschofield, Erxnmedia, Ethan, Eupedia, Freakofnurture, Fredrik, Furrykef, FvdP, Gaius Cornelius, Gene.arboit, Giftlite, Gnorthup, Grafen, Guahnala, Guslacerda, Hgranqvist, Hike395, HorsePunchKid, Huggie, Hyad, Imran, Intgr, JaGa, Jackzhp, Jaredwf, Jeff Edmonds, Jirislaby, Jiuguang Wang, Jmeppley, JonH, Julesd, Justin W Smith, Kaeslin, Karl-Henner, Ketil, Kku, LX, LachlanA, Leonard G., LiDaobing, Mahlon, Mark T, Matforddavid, Mdd, Meonkeys, Miaow Miaow, Mikeblas, Miss Madeline, Miym, Mlpkr, MrGBug, Mwj, Nils Grimsmo, Nowhere man, Npansare, Oleg Alexandrov, Oskar Sigvardsson, Paddu, Patrick O'Leary, Pekrau, Phatsphere, Philip Trueman, Pixiefeet, Pjrm, Pm215, Popnose, Qz, R'n'B, Rajkumar.p84, Richienumnum, Rinconsoleao, RustyWP, SSJemmett, Sam Hocevar, SamIAmNot, SavantEdge, SeldonPlan, Shantavira, Shuroo, Signalhead, Smmurphy, Spinmeister, Spireguy, Spmeyn, Sriganeshs, Stannered, Sydneyfong, Szarka, Tamfang, Terrifictriffid, The Thing That Should Not Be, TheMandarin, Tom Duff, Tommy2010, Tonya49, TripleF, Utcursch, VKokielov, Vecter, Vegpuff, Waxmop, Wonglijie, Zahlentheorie, Zarei.h, Zhouhowe, Ztobor, Zzyzx11, 297 anonymous edits

Linear programming *Source*: http://en.wikipedia.org/w/index.php?oldid=358692802 *Contributors*: .:Ajvol:., AJim, Abovechief, Ajgorhoe, Akasha27, Almi, Alotau, Andre Engels, Andreas Fabri, Andris, Apdevries, Arie ten Cate, Arjuna-vallabha, Artier, Ash211, Asm4, Barak, BarryList, Batpox, Bender2k14, Bengalftbl55, Bird of paradox, Bobblewik, Bobo192, Borgx, CRGreathouse, Caelumluna, Caesura, Charvest, ChaudhryZafar, Chemuser, Chris the speller, Chrislk02, Cjohnson, Ckhung, Cointyro, Cremepuff222, Crisgh, Ctbolt, Cybercobra, Czyl, DSP-user, DSachan, Dakee, Daveagp, Davekpeck, David.Monniaux, Dcoetzee, Ddechene, Delirium, Dianegarey, Dmitrey, Doobliebop, Doyley, Dpv, Dspoerl, Dysprosia, EdJohnston, Edadk, El C, EmmetCaulfield, Endersdouble, Erniepan, Falcon8765, Faturita, Fox, Fredrik, Freedomlinux, Freeman77, Fschoenm, Func, Fyyer, G.de.Lange, Gcm, Ghettoblaster, Giftlite, Gilliam, Gms, Gnasher729, Gshaham, Gwernol, Hairy Dude, Hgreenbe, HiYoSilver01, Hike395, Hjjalal, Hu12, Ike9898, Inductiveload, Isaacto, It Is Me Here, JRSP, JYolkowski, Jacj, Jakob.scholbach, Jhausauer, Jitse Niesen, Johnflux, Johngcarlsson, JonMcLoone, Joriki, JoshuaZ, Jpkotta, Jugander, Junglecat, Justforasecond, Justin W Smith, Jwestbrook, Kbdank71, Keilana, Kiefer.Wolfowitz, KnowledgeOfSelf, LachlanA, Ladislav the Posthumous, Lancedgang, Lavaka, LilHelpa, Lissajous, Lobizón, LorisRomito, Lradrama, Lschrage, Macintosh User, Marco Krohn, Marcol, Matforddavid, Mcmlxxxi, Mdd, Medvall, Michael Hardy, Michal Jurosz, Misfeldt, Misof, Miym, Mlpkr, Mmortal03, MrOllie, Msh210, Myleslong, Natebrix, NathanHurst, Nbarth, Nelson50, NeoChaosX, New Thought, Nick, Nigholith, Nimur, Nkhanna, Numsgil, Obradovic Goran, Oleg Alexandrov, Oliphaunt, Optimalon, Orderud, Oskar Sigvardsson, PMajer, Patrick, Pfortuny, Piano non troppo, PimBeers, Pkbmx, Qonnec, Quinnculver, RainbowOfLight, Rgrimson, Riedel, Rinconsoleao, RobinK, Robinh, Rsandvik, Saf37, Schmausschmaus, Schutz, Sdr, Sergnov, Shahab, SimpleKFC, Smokedsalmon crispyduck, Soliloquial, Spacepotato, Spalding, Splintercellguy, Steeldragon24680, Stu Mitchell, Tamfang, That Guy, From That Show!, The Anome, The Thing That Should Not Be, Thiseye, Thrufir, Thv, Tmkly3, Tobias Bergemann, Tohd8BohaithuGh1, Twocs, Twri, Ulric1313, Undsoweiter, Urdutext, Viridian, Vkuncak, Vocaro, Voltteri, Wastingmytime, Ylloh, Zheric, Zido, 438 anonymous edits

Time complexity *Source*: http://en.wikipedia.org/w/index.php?oldid=357045149 *Contributors*: Altenmann, CRGreathouse, Charles Matthews, Cybercobra, Eldar, Maxal, Michael Hardy, Miym, Mjoyce3012, RobinK, Romatt, Rwalker, Svick, WikHead, Ylloh, 7 anonymous edits

NP (complexity) *Source*: http://en.wikipedia.org/w/index.php?oldid=359734909 *Contributors*: 64.105.27.xxx, Abovechief, Ahmed saeed, Andris, AndyKali, Arvindn, Ascánder, Avkulkarni, AxelBoldt, AySz88, Bender2k14, Blokhead, Bones0, Catslash, Cheezycrust, Clecio, Conversion script, Creidieki, Dcoetzee, Dmr2, Docu, Duncancumming, Eijkhout, Falcone, Furrykef, Gdr, Giftlite, Giraffedata, GromXXVII, Gulliveig, Head, Helohe, Ikemccaslin, Jacobolus, Jan Hidders, Jao, Jasonwlh314, Jcarroll, Kaiserb, Kinkydarkbird, Kissedsmiley, Longhair, Marozols, Mellum, Michael Hardy, Mikaey, Miym, Mormegil, Nixdorf, Obradovic Goran, Oddity-, Ogai, Oliphaunt, OmriSegal, ParotWise, Perry Bebbington, Poor Yorick, Qwertyus, Rnsanchez, RobinK, Roentgenium111, Shreevatsa, Spiritia, Tarotcards, TehKeg, Template namespace initialisation script, That Guy, From That Show!, Timwi, Trovatore, Tsiaojian lee, Twri, Tylerl, Veganfanatic, Wap, Whkoh, William Ortiz, Wombleme, WuTheFWasThat, Ylloh, Zeno Gantner, 97 anonymous edits

Optimization (mathematics) *Source*: http://en.wikipedia.org/w/index.php?oldid=359822996 *Contributors*: APH, Aaronbrick, Ablevy, Ajgorhoe, Alphachimp, AnAj, Andris, Anonymous Dissident, Antonielly, Ap, Armehrabian, Arnab das, Arthur Rubin, Artur adib, Asadi1978, Ashburyjohn, Asm4, Awaterl, AxelBoldt, BenFrantzDale, BlakeJRiley, Bonnans, Bpadmakumar, Bradgib, Brianboonstra, Burgaz, Carbaholic, Carbo1200, Carlo.milanesi, Cfg1777, Chan siuman, Chaos, Charles Matthews, CharlesGillingham, Charlesreid1, ConstantLearner, Crisgh, Ct529, Czenek, DRHagen, Damian Yerrick, Daniel Dickman, Daryakav, Dattorro, Daveagp, David Martland, David.Monniaux, Deeptrivia, Delaszk, Deuxoursendormis, Dianegarey, Diego Moya, Diracula, Dmitrey, Doobliebop, Dpbert, Dsz4, Dysprosia, Ekojnoekoj, EncMstr, Encyclops, Erkan Yilmaz, Fred Bauder, G.de.Lange, Galoubet, Gglockner, Ggpauly, Giftlite, H Padleckas, H.ehsaan, HiYoSilver01, Hike395, Hosseininassab, Hu12, Hua001, Iknowyourider, Ish ishwar, Isnow, JFPuget, Jackzhp, Jasonb05, Jitse Niesen, Johngcarlsson, JonMcLoone, Jonnat, Jurgen, Justin W Smith, KaHa242, Kamitsaha, Karada, Klochkov.ivan, Knillinux, KrakatoaKatie, LSpring, LastChanceToBe, Lavaka, Lethe, Ltk, Lycurgus, Lylenorton, MIT Trekkie, Mange01, Mangogirl2, Marcol, MarkSweep, MartinDK, Martynas Patasius, Mat cross, MaxSem, MaximizeMinimize, Mcmlxxxi, Mdd, Mdwang, Michael Hardy, Mikewax, Misfeldt, Moink, MrOllie, Mrbynum, Msh210, Mxn, Myleslong, Nacopt, Nimur, Nwbeeson, Obradovic Goran, Ojigiri, Oleg Alexandrov, Olegalexandrov, Oli Filth, Optiy, Oğuz Ergin, Paolo.dL, Patrick, Peterlin, Philip Trueman, PhotoBox, PimBeers, Pohta ce-am pohtit, Polar Bear, Pontus, Pownuk, Procellarum, Pschaus, RKUrsem, Rade Kutil, Ravelite, Rbdevore, Remi Arntzen, Retired username, Riedel, Rinconsoleao, Roleplayer, Ryguasu, Sabamo, Salix alba, Sapphic, Saraedum, Schaber, Schlitz4U, Skifreak, Sliders06, Smartcat, Smmurphy, Srinnath, Stebulus, Struway, Suegerman, TPlantenga, Tbbooher, TeaDrinker, The Anome, Thoughtfire, Tizio, Topbanana, Travis.a.buckingham, Truecobb, Tsirel, Twocs, Van helsing, Vermorel, VictorAnyakin, Voyevoda, Wikibuki, Wmahan, X7q, Xprime, YuriyMikhaylovskiy, Zundark, Zwgeem, Щерол, 268 anonymous edits

P (complexity) *Source*: http://en.wikipedia.org/w/index.php?oldid=347847004 *Contributors*: Adam Zivner, Adiel, CRGreathouse, Creidieki, Dcoetzee, Esqg, Gdr, Giftlite, GromXXVII, Illuminatedwax, Kraven007mega, Markus Krötzsch, Michael Hardy, Mikaey, Neilc, NotQuiteEXPComplete, OmriSegal, Ott2, PaulTanenbaum, RobinK, SolifyDolphin, Stevertigo, Stokkink, TOGASHI Jin, That Guy, From That Show!, Trovatore, Twri, Uzyel, Visor, Wmahan, Ylloh, 23 anonymous edits

P versus NP problem *Source*: http://en.wikipedia.org/w/index.php?oldid=359900154 *Contributors*: 128.138.87.xxx, 194.117.133.xxx, 62.202.117.xxx, A bit iffy, Action Jackson IV, Adam McMaster, Adityad, Alexwatson, Alksentrs, Altenmann, Andejons, Andreas Kaufmann, Andris, Anog, Anonymousacademic, Anonymousone2, Archibald Fitzchesterfield, Arichnad, Arthur Frayn, Arthur Rubin, Arvindn, Aseld, Asmeurer, AstroNomer, AxelBoldt, Azimuth1, Banus, Bdesham, Bender235, Bigtimepeace, Bihco, Blazotron, Blokhead, Blotwell, Bofa, Booyabazooka, Brianjd, Brighterorange, C S, CALR, CBKAtTopsails, CRGreathouse, Calculuslover, Calvin 1998, Canon, Charles Matthews, Chinju, Chocolateboy, Chrisjwmartin, Civil Engineer III, CloudNine, Cngoulimis, Cointyro, Cole Kitchen, Conversion script, Creidieki, Cybercobra, Cyde, Damian Yerrick, DanielMayer, Dantheox, David Eppstein, David Gerard, David.Monniaux, Davidhorman, Dcoetzee, Deadcode, Debresser, Delmet, Dillon256, Dissident, Dlakavi, Docu, Donarreiskoffer, Doradus, Doulos Christos, Download, Dspart, Duncan, Duncancumming, ESkog, Escardo, Elizabeyth, EmersonLowry, EmilJ, Emurphy42, Eric119, Favonian, Fcady2007, Frazzydee, Fredrik, Gandalf61, Gary King, Gdr, Giftlite, Gonzolito, Graham87, GromXXVII, Gtcostello, Hairy Dude, Hcsradek, Hideyuki, Hobophobe, Hofingerandi, Hritcu, Ianhowlett, Icairns, Ihope127, Illuminatedwax, Iman.saleh, Intgr, IvanAndreevich, Jacob grace, Jammycakes, Jan

Hidders, Jaybuffington, Jitse Niesen, Jmah, Jon Awbrey, Jonnty, Jossi, Jtwdog, Julesd, Justin Stafford, Justin W Smith, Kralizec!, Kvikram, Kzzl, LC, LOL, Lambiam, Landroo, Laurusnobilis, LeoNomis, Libertyrights, Liron00, LouScheffer, Lowellian, Luckas Blade, Lukeh1, MER-C, Marknau, Markvs88, MartinMusatov, Mathiastck, MatthewH, Mattiabona, Mcsee, Mellum, Mentifisto, Mgiganteus1, Mgraham831, Michael Hardy, Mikaey, Minesweeper, Miym, Mpeisenbr, Ms2ger, Msikma, Najoj, Navigatr85, Nbhatla, Neutrality, Nicolaennio, Night Gyr, Nith Sahor, NuclearWarfare, Obradovic Goran, Oddity-, Olivier, Osias, Otisjimmy1, Paddu, ParotWise, Phil Boswell, Phil Sandifer, Pichpich, Pmadrid, Pohta ce-am pohtit, PoolGuy, Poor Yorick, Predawn, Qwertyus, R6144, RTC, Raiden09, Raven4x4x, Rbarreira, Rdnk, Remi Arntzen, Remy B, Rhythm, Rich Farmbrough, Rick Block, Rjwilmsi, Robbar, Robert Merkel, RobinK, Robinh, Rockiesfan19, Rorro, RoySmith, Rs561, Rspeer, Rtc, Rupert Clayton, Ruud Koot, Ryan Postlethwaite, Sabb0ur, Salgueiro, Schneelocke, Shawn comes, Shreevatsa, Sith Lord 13, Srinivasasha, Staecker, Stardust8212, Stephen B Streater, Stifle, Stuart P. Bentley, Subtilior, Sundar, Suruena, Svnsvn, Tcotco, Telekid, Template namespace initialisation script, Teorth, The Obfuscator, The Thing That Should Not Be, Thehotelambush, Thorbjørn Ravn Andersen, Thore, Timc, Timwi, Tizio, Tobias Bergemann, Trovatore, Ttonyb1, Twri, UkPaolo, Ultimus, Ustadny, Vadim Makarov, Versus22, Waldir, Wikiklrsc, Wikipediatrist, Wmahan, Wtt, Xiaodai, Ylloh, ^demon, 426 anonymous edits

NP-hard *Source*: http://en.wikipedia.org/w/index.php?oldid=359193833 *Contributors*: 151.99.218.xxx, 64.105.26.xxx, Agaib, Amelio Vázquez, Andris, Argumzio, Arthur Frayn, Arvindn, Ascánder, AxelBoldt, Beefman, Behnam, BenFrantzDale, Caesura, Cheezykins, Christopher Parham, Conversion script, Creidieki, Dmaciej, Donarreiskoffer, Drsteve, Duncancumming, Emj, Fcady2007, Gdr, Giftlite, Graham87, Gregbard, Hannes Hirzel, Ixfd64, Jafet, Jan Hidders, Jimbreed, Jonathan de Boyne Pollard, LC, Laurusnobilis, LizardWizard, Maksim-e, Martious, Mellum, Michael Hardy, Mikeblas, Mirer, Miym, Neilc, Obradovic Goran, Obscurans, Od Mishehu, Oliphaunt, Pgr94, Poor Yorick, Porqin, Ragzouken, Reddi, RobinK, Shreevatsa, Tarotcards, Template namespace initialisation script, Theone256, Tony1, Twri, Uriber, Wik, 81 anonymous edits

Computational complexity theory *Source*: http://en.wikipedia.org/w/index.php?oldid=359407265 *Contributors*: 137.112.129.xxx, 16@r, 24.93.53.xxx, 62.202.117.xxx, 64.105.27.xxx, APH, Abatasigh, Adavidb, Alotau, Altenmann, Andrei Stroe, Andris, Aphaia, ArnoldReinhold, Arthaey, Arvindn, Ascánder, Auminski, AvicAWB, AxelBoldt, Barcex, Bassbonerocks, Battamer, Beland, Ben Standeven, Bethnim, Bkell, Blokhead, Bo Jacoby, Booyabazooka, Bouke, Bruno Unna, Bsotomay, C. lorenz, CRGreathouse, Calculuslover800, Cesarsorm, Chalst, Charles Matthews, CharlesGillingham, Charvest, Chealer, Chinju, Christer.berg, Cngoulimis, ConceptExp, Contrasedative, Conversion script, Creidieki, D climacus, D.scain.farenzena, David Gerard, David Newton, David.Monniaux, DavidSJ, Dcoetzee, Decrease789, Deflagg, DerGraph, Dissident, Dmcq, Dmitri pavlov, Dmyersturnbull, Docu, Doradus, Drizzd, Droll, Déjà Vu, E23, Egriffin, Ehsan, Ekotkie, Epbr123, Erudecorp, Eser.aygun, Everyking, Flammifer, Four Dog Night, Fredrik, Fuujuhi, GPhilip, Gdr, Getonyourfeet, Giftlite, GrEp, Graham87, GregorB, Groupthink, Grsbmd, GulDan, Harryboyles, Hegariz, Henning Makholm, Henrygb, Hermel, Hfastedge, Hiihammuk, Hmonroe, Ink-Jetty, Intgr, Ixfd64, JRSpriggs, Jaksmata, Jamesd9007, Jeff Dahl, Jimbreed, Jitse Niesen, Jleedev, Jlpinar83, Johnuniq, Klausness, Knutux, Konstable, Kurykh, LC, Larry laptop, Leibniz, Linas, Little_guru, Looxix, Magmi, Manway, MathMartin, Mav, McKay, Mdd, Michael Hardy, Mik01aj, Mikeblas, Miym, Mpatel, Multipundit, Mycer1nus, N12345n, Nixdorf, Nneonneo, Obradovic Goran, Oleg Alexandrov, Omicronpersei8, OrgasGirl, Orz, Pascal.Tesson, Philip Trueman, Pichpich, Pohta ce-am pohtit, Populus, Postrach, Powo, PrologFan, Prumpf, Quackor, Quotient group, RainR, Readams, Rend, RexNL, Rich Farmbrough, Ripper234, Robert Merkel, RobinK, Ruud Koot, Ryguasu, Scottcraig, Shenme, Siddhant, SimonTrew, Skippydo, SpaceMoose, Stevenmitchell, Stevertigo, Talldean, Tarotcards, Tdgs, Tejas81, Template namespace initialisation script, The Anome, The Thing That Should Not Be, Tim32, Timwi, Tkgd2007, Tobias Bergemann, Triwas, Trovatore, Twri, Van Parunak, VictorAnyakin, Walkerma, Waltnmi, Wavelength, Wernher, WikHead, WikiSlasher, Wikiklrsc, Wvbailey, Xiaoyang, Yill577, Youandme, Young Pioneer, Zipcube, 182 anonymous edits

Transportation theory *Source*: http://en.wikipedia.org/w/index.php?oldid=354159906 *Contributors*: A. Pichler, Eastlaw, Geometry guy, Grin MD, Harryboyles, John Quiggin, Lambiam, Michael Hardy, Nguyen Thanh Quang, PranksterTurtle, Skapur, Sullivan.t.j, Zundark, Михајло Анђелковић, 10 anonymous edits

Image Sources, Licenses and Contributors

Image:TSP Deutschland 3.png *Source*: http://en.wikipedia.org/w/index.php?title=File:TSP_Deutschland_3.png *License*: Public Domain *Contributors*: Amirki, DanTD, Martynas Patasius, TUBS

Image:WilliamRowanHamilton.jpeg *Source*: http://en.wikipedia.org/w/index.php?title=File:WilliamRowanHamilton.jpeg *License*: Public Domain *Contributors*: unknown

Image:Weighted K4.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:Weighted_K4.svg *License*: Creative Commons Attribution-Sharealike 2.5 *Contributors*: Sdo

Image:Aco TSP.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:Aco_TSP.svg *License*: GNU Free Documentation License *Contributors*: User:Nojhan, User:Nojhan

Image:Knapsack.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:Knapsack.svg *License*: Creative Commons Attribution-Sharealike 2.5 *Contributors*: User:Dake

Image:Nonlinear programming jaredwf.png *Source*: http://en.wikipedia.org/w/index.php?title=File:Nonlinear_programming_jaredwf.png *License*: Public Domain *Contributors*: Jaredwf

Image:Nonlinear programming 3D.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:Nonlinear_programming_3D.svg *License*: Public Domain *Contributors*: User:McSush

Image:Decision_Problem.png *Source*: http://en.wikipedia.org/w/index.php?title=File:Decision_Problem.png *License*: GNU Free Documentation License *Contributors*: User:RobinK

Image:LampFlowchart.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:LampFlowchart.svg *License*: GNU Free Documentation License *Contributors*: User:Booyabazooka

File:Sorting quicksort anim.gif *Source*: http://en.wikipedia.org/w/index.php?title=File:Sorting_quicksort_anim.gif *License*: Creative Commons Attribution-Sharealike 2.5 *Contributors*: Berrucomons, Cecil, Chamie, Davepape, Diego pmc, Editor at Large, German, Gorgo, Howcheng, Jago84, Lokal Profil, MaBoehm, Minisarm, Miya, Mywood, NH, PatríciaR, Qyd, Soroush83, Stefeck, Str4nd, 11 anonymous edits

Image:PD-icon.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:PD-icon.svg *License*: Public Domain *Contributors*: User:Duesentrieb, User:Rfl

Image:Greedy algorithm 36 cents.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:Greedy_algorithm_36_cents.svg *License*: Public Domain *Contributors*: User:Nandhp

Image:Simplex description.png *Source*: http://en.wikipedia.org/w/index.php?title=File:Simplex_description.png *License*: Public Domain *Contributors*: Czupirek, Kocur, Maksim, Martynas Patasius, 1 anonymous edits

Image:Pert chart colored.gif *Source*: http://en.wikipedia.org/w/index.php?title=File:Pert_chart_colored.gif *License*: Public Domain *Contributors*: Original uploader was Jeremykemp at en.wikipedia

Image:Monte carlo method.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:Monte_carlo_method.svg *License*: Public Domain *Contributors*: --pbroks13talk? Original uploader was Pbroks13 at en.wikipedia

Image:Shortest path optimal substructure.png *Source*: http://en.wikipedia.org/w/index.php?title=File:Shortest_path_optimal_substructure.png *License*: Public Domain *Contributors*: User:Deco

Image:Fibonacci dynamic programming.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:Fibonacci_dynamic_programming.svg *License*: Public Domain *Contributors*: User:Stannered

Image:Linear Programming Feasible Region.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:Linear_Programming_Feasible_Region.svg *License*: Public Domain *Contributors*: User:Inductiveload

Image:Complexity classes.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:Complexity_classes.svg *License*: Public Domain *Contributors*: Booyabazooka, Mdd, Mike1024, 1 anonymous edits

File:MaximumParaboloid.png *Source*: http://en.wikipedia.org/w/index.php?title=File:MaximumParaboloid.png *License*: GNU Free Documentation License *Contributors*: Original uploader was Sam Derbyshire at en.wikipedia

Image:KnapsackEmpComplexity.GIF *Source*: http://en.wikipedia.org/w/index.php?title=File:KnapsackEmpComplexity.GIF *License*: Public Domain *Contributors*: User:Cngoulimis

Image:P np np-complete np-hard.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:P_np_np-complete_np-hard.svg *License*: GNU Free Documentation License *Contributors*: Original uploader was Behnam at en.wikipedia

Image:Decision Problem.png *Source*: http://en.wikipedia.org/w/index.php?title=File:Decision_Problem.png *License*: GNU Free Documentation License *Contributors*: User:RobinK

Image:Maquina.png *Source*: http://en.wikipedia.org/w/index.php?title=File:Maquina.png *License*: Public Domain *Contributors*: User:Schadel

Image:Complexity subsets pspace.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:Complexity_subsets_pspace.svg *License*: Public Domain *Contributors*: User:Qef

Image:Theoretical computer science.svg *Source*: http://en.wikipedia.org/w/index.php?title=File:Theoretical_computer_science.svg *License*: GNU Free Documentation License *Contributors*: User:RobinK

License