CS2309 CS Research Methodology

Problem Formulation

Lee Wee Sun
School of Computing
National University of Singapore
leews@comp.nus.edu.sg

Semester 1, 2011/12

Outline

- Principle of Simplicity
- Formulating application problems.
  - Abstract Models Principle
  - Principle of Generality
- Formulating systems problems.
  - Principle of Modularity
  - Information Hiding Principle
  - End-to-End Principle
- Formulating theory problems.
  - Worst Case Principle
  - Asymptotic Principle
  - Comparative Principle

Principle of Simplicity

Shannon

Suppose that you are given a problem to solve. I don't care what kind of a problem - a machine to design, or a physical theory to develop, or a mathematical theorem to prove, or something of that kind - probably a very powerful approach to this is to attempt to eliminate everything from the problem except the essentials; that is, cut it down to size. Almost every problem that you come across is befuddled with all kinds of extraneous data of one sort or another; and if you can bring this problem down to the main issues, you can see more clearly what you are trying to do and perhaps find a solution. Now, in do doing, you may have stripped away the problem that you're after. You may have simplified it to a point that it doesn't even resemble the problem that you stated with; but very often if you can solve this simple problem, you can add refinements to the solution of this until you get back to the solution of the one you started with.

Principle of Simplicity

Many notions of simplicity

- Small instances are simpler.
- Look at large instances but consider aggregate effects such as averages.
- Removing less important variables.

We will start with a simple example to illustrate the effect of removing some irrelevant details.

Knight Switch

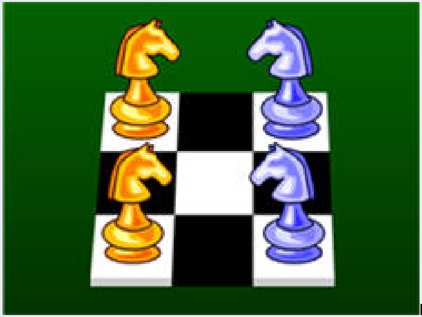
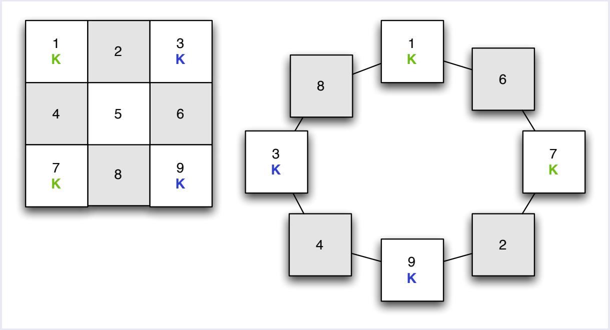


Image from

http://www.schooltimegames.com/Game_Art/Knight_Switch.jpg

- Switch the position of the knights of different colours.
- Only legal knight moves in chess allowed.



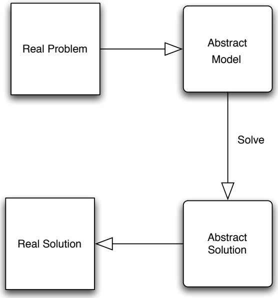
- Join two squares with an edge if there is legal move between them.
- Only essential detail represented.
  - Physical board not important.
- Easy to see the correct moves now.

Outline

- Principle of Simplicity
- Formulating application problems.
  - Abstract Models Principle
  - Principle of Generality
- Formulating systems problems.
  - Principle of Modularity
  - Information Hiding Principle
  - End-to-End Principle
- Formulating theory problems.
  - Worst Case Principle
  - Asymptotic Principle
  - Comparative Principle

Abstract Models Principle

- Once unnecessary details are removed, we often find that the abstract problem is the same as an abstract problem that we have solved before.
- No need to solve a new problem!



- This suggests explicitly looking to *reduce* a problem into a known abstract model.

- Need lots of knowledge on useful abstract models.
- In this course, we will look at
    - *graphs* and
    - *propositional logic*

  as two powerful abstract models to illustrate this principle and other principles later in the course.

Graphs

- A graph consist of a set of vertices V and a set of edges E that represent binary relations between the vertices.
- It is very useful for abstracting out only essential relationships between objects.
- Look at reduction of practical problems into a few commonly studied graph problems.
    - Independent set, Dominating set
    - Coloring
    - Shortest path
    - Shortest and longest path in a DAG
    - Traveling salesman, Spanning tree

Eight Queens

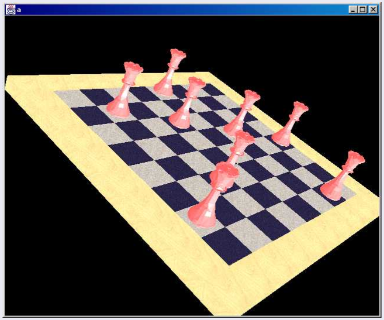


Image from `http://research.microsoft.com/~nick/scrshot.jpg`

Place eight queens on a chess board such that no queen can attack another. Represent this as an abstract graph problem.

Independent Set

- Terminology
    - Two vertices u and v are adjacent if there is an edge $(u, v) \in E$ between them.
    - An independent set I is a subset of V such that no two vertices in I are adjacent.
- Often interested in maximum independent set - largest such set.

What is the eight queens graph?

Class Feedback

- There are *n* students in the CS2309 class.
- Some of the students know each other. To get feedback about the course, the teaching staff want to talk to a subset of students.
- The subset must be selected so that every student knows (and is known by) someone in the subset.
- To save time, the teaching staff want to find the smallest such subset.

Dominating Set

- Assign a vertex to each student.
- Join two vertices by an edge if the students know each other.
- A vertex dominate itself and any adjacent vertex.
- A dominating set is a set of vertices S such that every vertex in the graph is dominated by some vertex in S.
- Find a minimum dominating set: a dominating set that contains the smallest number of vertices.

Map Colouring

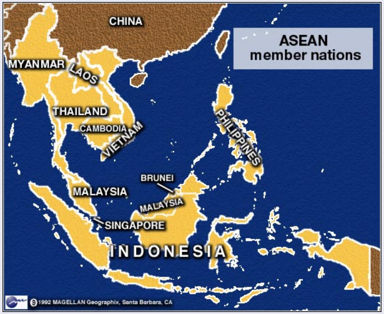


Image from `http://www.cnn.com/WORLD/asiapcf/9812/15/asean.summit/ASEAN.countries.map.jpg`

Given a map of countries, color the map using the smallest number of colors such that any two countries that share a border are not colored the same color. Represent this as an abstract graph problem.

Graph Colouring

- Vertices represent countries.
- Join any two countries that share a border with an edge.
- A coloring of a graph G is an assignment of the vertices in the graph G so that any two vertices connected with an edge have different colours
- The optimum graph colouring problem is to find a coloring of the graph that uses the fewest colors.

Road Map

Image from `http://www.popular.com.sg/images/product/book/69101.jpg`

We want to find the shortest path between one town to another assuming that the length of each road on the map is specified by a label next to it. Represent as an abstract graph problem.

Shortest Path

- In a directed graph (digraph), the edge (u, v) where u and v are vertices is different from the edge (v, u).
- In a weighted graph, a real number w called the weight is associated with each edge.
- A path from a vertex u to a vertex v is a sequence $\langle v_0, v_1, \ldots, v_k \rangle$ of vertices such that $u = v_0$, $v = v_k$, (v_{i-1}, v_i) are members of E and all the vertices are distinct.
- The weight of a path is the sum of the weights of its constituent edges.
- Find a path from the source to target vertex with the least weight.

Represent the Road Map problem as a shortest path problem.

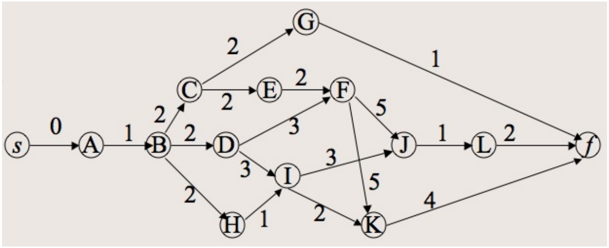
Car Assembly

- In an automobile assembly plant the work is broken down into a number of tasks, for each of which the time needed is known.
- The physical circumstances impose certain ordering relations among the tasks, that is, some cannot be started until others have been completed.

Task	Code	Time	Must follow
Body Panel	A	1	-
Chasis	B	2	A
Engine	C	2	B
Transmission	D	3	B
Steering	E	2	C
Electric	F	5	D, E
Wheels	G	1	C
Doors	H	1	B
Painting	I	2	D, H
Windows	J	2	F, I
Interior	K	4	F, I
Test	L	2	J

What is the minimum time needed to complete an assembly, assuming that enough manpower is available to perform tasks in parallel whenever necessary?

Longest Path in a DAG



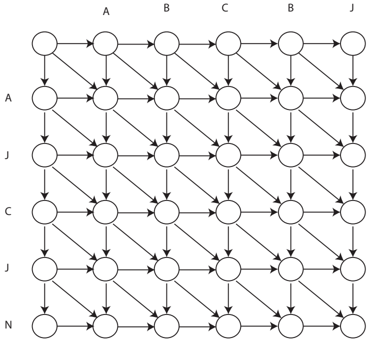
- Each vertex corresponds to a task.
- Two extra vertices s, f have been added for the start and completion of the task.
- Two vertices u and v are linked with a directed edge from u to v if u must be completed before v can be started.
- The edges leaving a node has weight associated with the time required to complete the task at the node.
- Find the longest path from s to f.

Distance between proteins

- In computational biology, we often want to find related genes or proteins.
- One simple abstraction is to assume that a protein is represented by its amino acid sequence. There are 20 types of amino acids, so a protein is just a string like "AJCJNR...".
- During evolution, proteins are often changed through mutation of one amino acid into another, insertion of amino acids or deletion of amino acids.
- A simple distance between sequences - assuming mutation, insertion and deletion have certain costs each, compute the cheapest way to transform one string into another.

Shortest Path in a DAG

- Use a simple two dimensional grid of vertices.
- Each row of vertices is labeled the source sequence ("AJCJN" in the example)
- Each column is labeled with the target sequence ("ABCBJ" in the example).
- Diagonal edge weights encode the cost of substituting the row symbol with the column symbol.
- Horizontal edge weights encode the cost of inserting the column symbol.



- Vertical edge weights encode the cost of deleting a row symbol.
- The task is to find the shortest path from the top left node (special start node) to the bottom right node.

Travelling Salesman

- If we relax the definition of a path to allow the first and last vertices to coincide, we call the resulting closed path a cycle.
- The travelling salesman problem is equivalent to finding the shortest cycle which goes through every vertex: a tour.

Travelling Salesman

- A salesman has n customers, all in different towns; he wished to visit them all in turn, starting and ending his journey at the same town and not passing twice through an intermediate town.
- Knowing the distances between every pair of towns, in what order should he make the visits so as to travel the least possible total distance?

Communication Network

We would like to build a communication network linking n cities. We know the cost of linking each pair of cities. It is possible to link all cities together using $n-1$ links. Find a way to link together all the cities at the minimum cost.

Minimum Spanning Tree

- Model the problem as an undirected weighted graph, where the set of cities form the vertices and an edge is formed between each pair of cities that can be connected by a direct link.
- The weight of the edge is the cost of linking together the two cities.
- We say that two vertices u and v are connected if there is a path from u to v.
- We wish to find a subset of edges which connects all the vertices and has the least total weight.
- Such a subset necessarily form a *tree* which we call a spanning tree since it spans the graph.
- We call the problem of finding the tree with the minimum cost the Minimum Spanning Tree problem.

Satisfiability

- There are many ways to represent the same problem.
- Different representation may have different advantages and provide different insights.
- Whether there is an assignment of variables to make a boolean formula true is known as the satisfiability problem.
- For example the formula $(x \vee \neg y) \wedge (\neg x \vee y)$ has a satisfying assignment $x = 1$ and $y = 1$.
- Formulas of the form used above (conjunctions of disjunctions where the disjunctions are known as clauses) are known as conjunctive normal form (CNF).

Colouring as CNF

- Is there a three colouring of this graph?
- One encoding would be to use three variables r_i, g_i, b_i for each node to represent colour. Encode conjunction of the following.
- Encode the constraints that each node has at most one colour: : $\neg[(r_i \wedge g_i) \vee (r_i \wedge b_i) \vee (b_i \wedge g_i)]$ giving

  $$(\neg r_i \vee \neg g_i) \wedge (\neg r_i \vee \neg b_i) \wedge (\neg b_i \vee \neg g_i) \text{ for each } i,$$

  using DeMorgan's rule.
- There are no uncolored nodes:

  $$(r_i \vee g_i \vee b_i) \text{ for each } i.$$

- Nodes connected by an edge must have different colors:

  $$(\neg r_i \vee \neg r_j) \wedge (\neg g_i \vee \neg g_j) \wedge (\neg b_i \vee \neg b_j) \text{ for each edge } (i, j).$$

The Right Level of Details

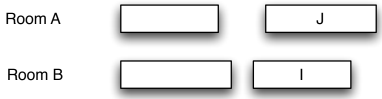
Minimize Rooms

- The School of Computing has n identical tutorial rooms.
- Given the tutorial timetable for the semester, it is desirable to use the fewest rooms possible so that the remaining rooms can be designated as student rooms for students to use in between their classes.
- How do you minimize the number of tutorial rooms used?

- Each vertex represents a class. Connect vertex i to vertex j with an undirected edge if the period for class i intersects with the period for class j.
- Find the smallest number of colors required to color the graph such that no two vertices connected by an edge share the same color.
- Is solving this using a general graph coloring solver a good idea?

- General coloring problems are NP-complete but this particular case (on intervals) has efficient algorithmic solutions.
- Simply sort the intervals by their starting time and process them according to the sorted order.
- For each interval (class), find a previously used room that is not currently used and assign it to the class - if all previously used rooms are currently used, add a new room.

- To see that this is optimal, consider an optimal assignment and the first time that the strategy differs in room assignment with the optimal assignment.
- In the optimal assignment, an interval J that starts later is assigned to the room A that would have been assigned to the current interval I by our strategy and I is assigned to room B



instead.
- We can construct another optimal assignment that agrees with our assignment for I simply by swapping all the intervals after and including I and J in the two rooms.
- This swapping process does not increase the number of rooms used and can be done because our interval starts the earliest among the remaining intervals (see figure).

Adding Details

- Abstracting out too much can cause loss of details that can make the problem easier to solve.
- However, often easier to abstract too much and add details (recall Shannon's advice).

- This swapping process can be done each time a disagreement appears until we obtain an optimal assignment that agrees with our assignment.
- Adding the detail that the vertices represent intervals allows us to use sorting instead of a colouring solver designed to solve an NP-complete problem.

Outline

- Principle of Simplicity
- Formulating application problems.
  - Abstract Models Principle
  - Principle of Generality
- Formulating systems problems.
  - Principle of Modularity
  - Information Hiding Principle
  - End-to-End Principle
- Formulating theory problems.
  - Worst Case Principle
  - Asymptotic Principle
  - Comparative Principle

Principle of Generality

- While solving the problem, you should always think about how to generalize the problem.
- The problem is usually an instance of a class of problems.
    - Extracting out the general properties of the problem may allow you to solve for a class rather than a specific problem.
    - Solving a more useful and important problem.

- Paradoxically, general sometimes means easier for human.
- Less details to distract human. E.g.
    - For protein similarity, resulting abstract edit-distance problem can be applied in many other applications: spelling correction, music similarity, etc.
    - Yet, the shortest path in the DAG problem easy to solve, even by hand.

Example: Google PageRank and Random Walk on Graphs

- Around 1998, Altavista was the dominant search engine
- Google came along and gave much better search result and became dominant
- Why? Much better model for ranking!
- Treat web as graph
    - Vertices as web pages
    - Directed edges are user annotated links from page to page
- Exploit human understanding through human annotated links
    - Much better compared to looking for presence of query words in document

Ranking criteria:

- Assume web surfer moves randomly from page to page
- At each time step, randomly select an link on current page and follow to next page
- Rank accordingly to probability of finding surfer on a page

Simple abstract model, simple criteria. Once defined this way, algorithm for solving (finding probability in random walk on graphs) already known (and simple)!

More details in paper reading.

Outline

- Principle of Simplicity
- Formulating application problems.
  - Abstract Models Principle
  - Principle of Generality
- Formulating systems problems.
  - Principle of Modularity
  - Information Hiding Principle
  - End-to-End Principle
- Formulating theory problems.
  - Worst Case Principle
  - Asymptotic Principle
  - Comparative Principle

Systems Problems

Issues that we are often concerned with:

- Correctness
- Efficiency
- Scalibility
- Reuse/Changes
- Reliability
- Security
- Real-time constraints

The following principles often help address some of these issues.

Principle of Modularity

Principle of modularity:

- The main idea is to break large systems into smaller parts so that the complexity of each part is small enough to cope with.
- If each part is small enough, it can be designed, understood and tested thoroughly to ensure correctness.

We give some commonly seen examples of application of this principle.

Layers of Abstraction

Computer Organization

- Computer systems are often designed using a layered architecture.
  - Resistors, capacitors, transistors, used to build gates
  - Logic gates used to build control and data paths of processor
  - Machine language using binary representation
  - Assembly language
  - Programming languages
  - Applications
- Easier to build, problem decomposed into building one layer at a time.
- With properly specified interface, can be done in parallel.
- Allows one level to change without affecting the higher level e.g. the hardware keeps improving but the same programs still run.

The *THE* Operating System

- Operating systems are often designed in layers.
- THE (Technische Hogeschool Eindhoven) is an influential early multiprogramming system designed by a team led by Edsger Dijkstra.
- Consist of 6 layers:
  - *Layer 0* was responsible for multiprogramming aspects including interrupts, context switches when changing processes, etc.
  - *Layer 1* was concerned with memory allocation to processes.
  - *Layer 2* handled inter-process communication and communication with the console.
  - *Layer 3* handled all I/O with devices connected to the computer.
  - *Layer 4* are the user programs.
  - *Layer 5* was the system operator which had overall control.

- Building the OS in layers allowed Dijkstra's team to design and test the system incrementally. When Layer n is built, it can assume that Layer $n-1$ is correct, reducing the testing required.

OSI Model

- Communication systems are usually modeled using 7 layers:
  1. Physical layer: Media, signal and binary transmission
  2. Data link: Physical addressing (Ethernet, etc.)
  3. Network: Logical addressing and paths (IP)
  4. Transport: End to end connection (TCP)
  5. Session: Interhost
  6. Presentation: data encoding (MIME, encryption)
  7. Application (file transfer, etc.)

Outline

- Principle of Simplicity
- Formulating application problems.
  - Abstract Models Principle
  - Principle of Generality
- Formulating systems problems.
  - Principle of Modularity
  - Information Hiding Principle
  - End-to-End Principle
- Formulating theory problems.
  - Worst Case Principle
  - Asymptotic Principle
  - Comparative Principle

Information Hiding Principle

- How to decompose a problem into modules?
- The commonly accepted criteria for decomposing problems advocated by Parnas is called information hiding.

Parnas (1972)

..it is almost always incorrect to begin the decomposition of a system into modules on the basis of a flowchart. We propose instead that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others.

In the classic book, *Mythical Man-Month*, Brooks called Parnas's approach "a recipe for disaster" and described his approach:

Brooks

In the Operating System/360 project, we decided that all programmers should see all the material i.e., each programmer having a copy of the project workbook, which came to number over 10,000 pages. Harlan Mills has argued persuasively that "programming should be a public process," that exposing all the work to everybody's gaze helps quality control, both by peer pressure to do things well and by peers actually spotting flaws and bugs.
This view contrasts sharply with David Parnas's teaching that modules of code should be encapsulated with well-defined interfaces, and that the interior of such a module should be the private property of its programmer, not discernible from outside. Programmers are most effective if shielded from, not exposed to, the innards of modules not their own.

Twenty years later, Brooks said "Parnas Was Right, and I Was Wrong about Information Hiding".

- Information hiding commonly used when implementing library data structures. Examples:
  - It is useful to use a set without worrying whether it is implemented by a list, a balanced binary search tree, a hash table etc.
  - A queue may be an array, a linked list, etc.
  - A priority queue may actually be implemented using a heap, a binary search tree, etc.
  - A graph may be represented using an adjacency matrix or adjacency list.

Data structures usually encapsulated together with the operations such as *insert*, *delete*, *update* as an *abstract data structure*.

- Type of encapsulation is facilitated by object-oriented programming
    - Programming language allows both the data and procedures associated with it to be hidden behind one interface.

Outline

- Principle of Simplicity
- Formulating application problems.
  - Abstract Models Principle
  - Principle of Generality
- Formulating systems problems.
  - Principle of Modularity
  - Information Hiding Principle
  - End-to-End Principle
- Formulating theory problems.
  - Worst Case Principle
  - Asymptotic Principle
  - Comparative Principle

End-to-End Principle

Where should we put a particular function, particularly in layered systems, such as those seen earlier?

File Transfer

Consider the problem of transfering a file from a computer A to another compute B. The following are the possible threats:

1. Reading from disk at A contains error.
2. Software at A or B makes a mistake in buffering and copying the data.
3. Hardware or local memory have transient error at A or B.
4. Data communication system between A and B corrupts the data.
5. Either A or B crash part way through the transaction.

How should the system deal with the threats to ensure reliability?

- One approach would be to reinforce each of the steps along the way
  - use duplicates, retries, error correction, etc.

  The goal is to reduce the probability of error along each step so that the final program is reliable.
- Another approach is a end-to-end check and retry method.
  - Each file is stored with a checksum.
  - After a file has been written to disk, it is read back into memory and the checksum is used to ensure that there is no error.
  - If there is error, a retry from the beginning is done.

Which of the two methods should be prefered?

- To ensure correctness, the end-to-end check will have to be done regardless of whether the extra effort has been done in the remaining steps.
- The effect of the other steps is to decrease the frequency of retries, not eliminate the need for it.
- If the error rate is small (and it usually is for these types of applications), building the system using the first method not only takes more work, it is also less efficient due to the overhead at each step.

End-to-End Principle

- The end-to-end principle tries to answer is where to place a particular functionality, particularly when there are many places where it could conceivably go.
- The principle suggests that many functionalities can be completely and correctly implemented only at the application level and placing them at low levels of a system may be inefficient and often redundant.
- A rough guide when building a system to be used by many users: If
  - uncertain whether users will use a function heavily, and
  - the function is expensive to implement and/or will slow down other frequently used functions

  Then leave it to the user to implement.

Example: RISC architecture

- The reduced instruction set computer (RISC) architecture tries to provide only the simplest instruction set and depends on higher level tools to construct complicated functionalities.
    - The computer designer is unlikely to be able to predict the instructions that will be most useful for any application
    - Optimizing the instruction set for one application is likely to lead to poorer performance on other applications.
- It is better to provide the simple but fast instructions
    - Leave the construction of complex functionalities from simple instructions to the application (usually compilers) which has the information to optimize better.

World Wide Web

- WWW is a notable failure of computer systems research - it was invented by a physicist. ☹
- One reason for the WWW's success is what Tim Berners-Lee calls the principle of least power: when choosing a representation, choose the least powerful one.
- HTML was chosen as language for the web because
  - Easy to use without a lot of training
  - Easy for users to do various things with it: presentation, indexing, extracting tables, etc.
- Related to end-to-end principle: When unclear what users will use the system for,
  - Keep it simple.
  - Make it efficient/easy to use.
  - The user knows best what they want to do with it.

Principle of Simplicity Redux

Simplicity may not always be so simple.

Relative Simplicity

- Simplicity may be relative.
- Something can be simple relative to what is known or expected - it does not convey or require new information.
- Also known as principle of least surprise, is particularly useful with respect to human expectation.
  - If you are designing a calculator, "+" should always refer to addition.
  - If there are usage conventions, you should always follow it.

Case Study: Relational Model

One of Codd's main motivation for proposing the relational database is its ability to hide the implementation of the database from the consumer of the data (information hiding principle).

A relational database represents a set of entities using a collection of relations (tables). Each table consists of

- rows that are unique but whose order is unimportant.
- columns that are named but whose ordering is unimportant.

Handling changes

- Compare a table with a hierachical (tree) representation.

Example:

Student	*Lecturer*	*Course*
Alice	Dr A	CS1101
Bob	Dr B	CS1102
...	...	...

- There are nine possible trees for representing this data: three each rooted by *Student*, *Lecturer* or *Course*.
- If we decide to change the root, all programs that use that database will have to be changed.
- In contrast, if we only allow other programs to know the definition of the relation (table) and access data through the database query and update routines, the database will still work regardless of change of implementation details, e.g. indexed using a balanced tree instead of a hash table, etc.

Query Language

- Relational algebra, consisting of operations such as selection, projection, join, union, difference and renaming gives a query language for the relational model.
- Output of an operation on a relation is also a relation (closed under the operations) - simple and elegant.
- But there are queries, even those that seems natural for a relational database that cannot be made in relational algebra e.g. finding the transitive closure of a relation.

- What to do with queries that cannot be made with relational algebra?
- If they are not common but would take substantial resources, the end-to-end principle suggests letting the application take care of them.
- Indeed, earlier versions of SQL implements a superset of relational algebra but did not allow transitive closure.

- The principle of simplicity suggests that an abstraction should be as simple as possible.
- However, functions that are central cannot be left out.
- One important characteristic of databases cannot be ignored - data is modified, inserted and deleted all the time.
- Throughout all these changes, the correctness of the data must be preserved.

Integrity Constraints

- To help preserve correctness in the data in face of changes, data integrity constraints is included in the relational model.
- Each table field is associated with a *domain constraint* (e.g. the type has to be integer).
- *Key* constraints specify subsets of fields that must be unique in a table.
- *Foreign key* constraints ensure that the value in a field must be present in another table.
- *Normalization* is a process that breaks up large tables into smaller tables to reduce problems cause by changes in the presence of redundant information.

Outline

- Principle of Simplicity
- Formulating application problems.
    - Abstract Models Principle
    - Principle of Generality
- Formulating systems problems.
    - Principle of Modularity
    - Information Hiding Principle
    - End-to-End Principle
- Formulating theory problems.
    - Worst Case Principle
    - Asymptotic Principle
    - Comparative Principle

Worst Case Principle

- The principle of simplicity applies very strongly for theory.
    - Even after abstracting away the unimportant details, analysis of problems and algorithms is often still difficult.
    - Sometimes simplicity prefered over realism, if simplicity provides insight.
- Worst-case is one of most common assumption.
    - Average case may be more useful, but usually distribution not known.
    - Worst cases provides guarantee, regardless of what happens.
- Example: worst-case runtime used to justify prefering merge sort over insertion sort.

Asymptotic Principle

- Analysis often becomes simpler when the number of elements is large.
  - Often only the rate of growth matters.
  - Example: Insertion sort (runtime $K_1 n^2$) vs merge sort (runtime $K_2 n \log n$), ignore the constants.
- In asymptotic notation, when we say that the running time is
  - $O(f(n))$ when there exists positive constants K. and n_0 such that the running time is less than $Kf(n)$ for all $n \geq n_0$.
  - $\Omega(f(n))$ when there exists positive constants K. and n_0 such that the running time is more than $Kf(n)$ for all $n \geq n_0$.
  - $\Theta(f(n))$ when the running time is both $O(f(n))$ and $\Omega(f(n))$.

Comparative Principle

Absolute values often do not provide useful insights; comparing the value against something else is often more insightful.

Competitive Analysis

- Asymptotic worst case analysis sometimes does not give the right insights.
  - Example, in paging, we can generate inputs that will cause every request to be a miss.
- Average case analysis is difficult as the distribution of problems unknown.
- Competitive analysis compares performance of the algorithm against the optimal performance on each instance of problem.

- An algorithm A is competitive if, for each input, its cost C_A satisfies $C_A \leq \alpha C_{\text{Opt}} + \beta$, where α, β are constants and C_{Opt} is the optimal cost for that input.
- If the problem instance is easy, C_{Opt} is small and C_A has to be correspondingly small. If the problem instance is difficult and C_{Opt} is large, then C_A can also be large.
- See Sleator and Tarjan's paper which analyses self-organizing list and paging using competitive analysis in paper reading.

Case Study: Theory of NP-Completeness

$P = NP$? The most famous, and arguably the most important problem in theoretical computer science.
Relationship to principles we looked at:

- Worst case principle: NP-completeness is worst case result. May not be right formulation for some problems, but nevertheless useful insight.
- Asymptotic principle: NP-completeness is asymptotic result. Useful for understanding scalability. If problem size is small, can still solve.
- Comparative principle: NP-complete means as difficult as the most difficult problem in NP. Useful even though we do not know the true running time.