Formulation

Having selected a research problem and determined that it will be impactful if solved, what next? You'd probably do some small exploratory studies first. Look at whatever data points that you can find. Look at some simple instances. Construct some simple examples that give good outcomes and maybe some simple examples that give poor outcomes.

Having gained some intuition, you usually need to formulate it to an extent that you can solve it meaningfully. This usually mean constructing a mathematical model, possibly with an optimization criterion. We will discuss some key things to look out for when formulation problems.

1 Principle of Simplicity

This principle applies in almost everything you are doing, whether it is an application, system, proving theorems, or formulating scientific hypotheses. Shannon said it very well [12]:

> *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.*

A side effect of removing details is that the problems become more *general*. So as a bonus, the solution often applies to a larger class of problems as well. Of course, simplicity is a slippery notion. A problem with a small number of instances can be considered simple - so you should almost always think about what happens when there is a small number of instances. However, it may be that it is simpler to think about the problem when it is very large and only consider aggregrate effects such as averages. Removing the number of variables that characterize the problem usually simplifies it, so try to remove variables that are unnecessary to solve the problem.

We will give a little example here to illustrate how removing details can simplify a problem enough to render it almost trivial, providing you with an "Aha!" insight.

Knight Switch: Switch the position of the knights of different colours. Only legal knight moves in chess allowed[1].

[1] Image from `http://www.schooltimegames.com/Game_Art/Knight_Switch.jpg`

This problem appears to be difficult to think about. However, if we remove the detail of the relative positions of the squares on the board and focus only on which square can be reached by a knight from the current square, the structure of the problem reveals itself. This is shown in Figure 1. Observe that the reachable squares has cyclic chain structure. Using the fact that no two knights can be placed on the same square at any time leads to the conclusion that the ordering of the knights on the chain cannot change. Therefore there are only two ways to move them to the new legal configuration where the knight of different colours have been switched: either move every knight in the clockwise manner, or move every knight in the anti-clockwise manner. Either way leads to 16 moves, giving 16 as the minimum number of moves required.

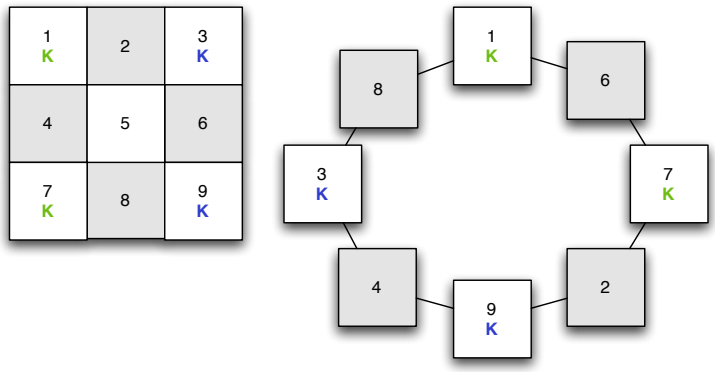


Figure 1: Connecting the squares that can be reached a knight from each position reveals the structure of the problem.

2 Formulation for Practical Applications

2.1 Abstract Models Principle

Once we remove unnecessary details, we may find that the abstract problem is the same as a well known one - no need to invent new method. So one effective problem solving method is to try to *reduce* the problem to a known abstract model. This method is illustrated in Figure 2. For this to

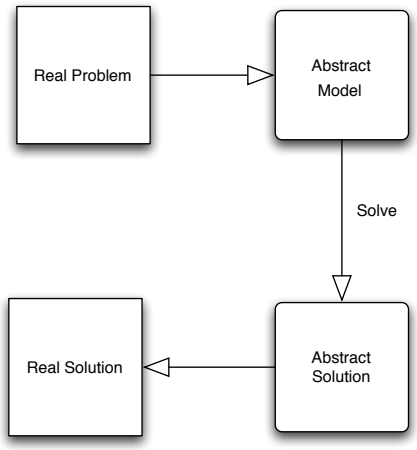


Figure 2: Solving a problem by reduction to known abstract model.

work well, you need to have a lot of knowledge, i.e. you need to have studied and mastered many abstract models.

We will look at two types of abstract models that are often useful in many computer science applications: using graphs and propositional logic. You will, of course, learn many other useful abstract models. But these two representations will be used as illustrations in this course[2].

2.1.1 Graphs

A graph consist of a set of vertices V and a set of edges E that represent binary relations between the vertices. Two vertices u and v in a graph are adjacent if there is an edge $(u, v) \in E$ between them. An edge (u, v) is incident to the vertices u and v.

With only vertices and edges, graphs are wonderful at removing irrelevant details. We will consider a few simple graph problems.

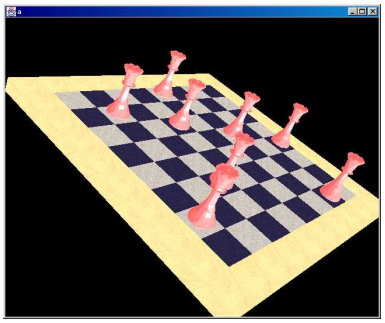
Independent Set

Eight Queens Problem: Place eight queens on a chess board such that no queen can attack one another[3].

Let's change this into a graph problem by removing unnecessary details. Use a vertex to represent a board position. Connect two vertices by an edge if the two corresponding board positions can be reached by a queen move (on the same row, column or diagonal). An *independent set*, I, is a subset of V such that no two vertices of I are adjacent. In this problem, we are interested in finding an independent set of size eight. Often, we are interested in finding the *maximum independent set*, the

[2]Mastering a few of the most useful representations will serve you well - the 80-20 rule says that 80% of the problems you encounter will be effectively solvable using 20% of the techniques.

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



largest possible independent set (eight for this problem). Notice how we have abstracted away the fact that there is a board and chess pieces.

Dominating Set

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.

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

Graph Colouring

Map Colouring: Given a map of countries, colour the map using the smallest number of colours such that any two countries that share a border are not coloured the same color[4].

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

Represent each country as a vertex. If one country share a border with another country, connect the corresponding vertices 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 colouring of the graph that uses the fewest colours.

Shortest Path

Road Map: 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[5].

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.

For the road map problem, we can represent each junction as a vertex. If junction v can be reached from junction u along a single road segment, add an edge (u, v) with a weight representing the segment length. The problem is then to find the shortest path from a vertex representing one town to another vertex representing another town.

Shortest/Longest Path in a DAG

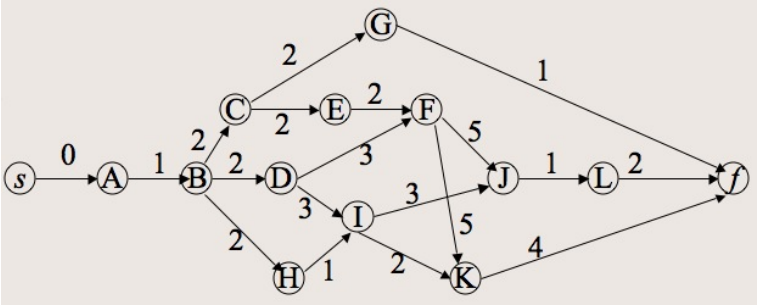
Car Assembly: In an automobile assembly plant the work is broken down into a number of tasks, for each of which the time needed is known. The physical circumstances impose certain ordering relations among the tasks, that is, some cannot be started until others have been completed. This is shown in Table 1. What is the minimum time needed to complete an assembly, assuming that enough manpower is available to perform tasks in parallel whenever necessary?

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. Note that this graph is a *directed acyclic graph (DAG)*,

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

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

Table 1: Ordering relations among tasks in car assembly.



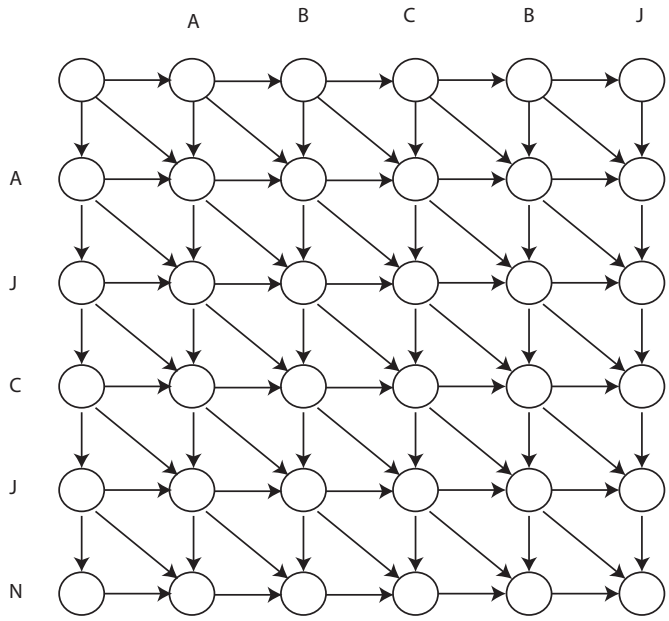
i.e. it is directed and has no cycles. A task cannot be started until all the tasks that it depends on has completed. Hence, completing all tasks must take at least as long as the weight of any path from s to f. Finding the longest path from s to f will give the shortest time possible to complete the task with unbounded resources.

Distance between proteins: In computational biology, we often want to find related genes or proteins. For example if we have a protein in human, we may want to find a similar protein in mouse. If the function of the mouse protein is well understood, this may be able to shed light on the human protein.

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...". Among other things, this abstraction does not retain the 3-D structure of the protein.

To find a similar sequence, we need to define a distance between sequences. Biologically, proteins of the same functions are often evolved from the same parent. During evolution, proteins are often changed through mutation of one amino acid into another, insertion of amino acids or deletion of amino acids. This motivates a simple distance between sequences

- assuming mutation, insertion and deletion have certain costs each, compute the cheapest way to transform one string into another [8][6].



We use a simple two dimensional grid of vertices. Each row of vertices is labeled the source sequence ("AJCJN" in the example) while 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 while 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

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?

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.

Minimum Spanning Tree

[6]This is a simplified version of the method presented by Needleman and Wunsch.

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.

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.

2.1.2 Satisfiability

There are many different abstract 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 (SAT) problem. For example the formula $(x \vee \neg y) \wedge (\neg x \vee y)$ has a satisfying assignment $x = T$ and $y = T$. Formulas of the form used above are known as conjunctive normal form (CNF). A CNF is a conjunction of clauses, where each clause is a disjunction. In the example above, the clauses are $(x \vee \neg y)$ and $(\neg x \vee y)$.

SAT played a pivotal role in helping computer scientists understand what problems can be solved quickly. It was the first example of an NP-complete problem. Decision problems whose solutions can be *verified* in polynomial time belong to the class NP. It is known that all problems in NP can be transformed into SAT in polynomial time. This means that if SAT can be solved in polynomial time, all problems whose solution can be verified in polynomial time can be solved in polynomial time. Most computer scientists believe that SAT cannot be solved in polynomial time. However, SAT solvers are surprisingly effective for many practical problems, so representing problems as SAT and using an effective SAT solver can be an effective problem solving method.

Colouring as CNF For simplicity, we only consider encoding for three colours i.e. given a graph G, is it three-colourable? One encoding would be to use three variables r_i, g_i, b_i for each node to represent colour, where r_i denote the proposition that node i has colour r. 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[7].
- There are no uncolored nodes:
$$(r_i \vee g_i \vee b_i) \text{ for each } i.$$

[7]DeMorgan's rule states that $\neg(a \vee b) \Leftrightarrow \neg a \wedge \neg b$.

- Nodes connected by an edge must have different colors:

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

Independent set as CNF We will encode for the question: Given a graph G, is there an independent set of size k? Assign a value from 0 to k to each node where 1 to k represent one of the vertices in the independent set each while 0 represent vertices not in the independent set. We can use $k + 1$ binary variables $0_i, 1_i, \ldots, k_i$ for each node.

- Each node is assigned at least one value: $(0_i \vee 1_i \ldots \vee k_i)$ for each i.
- Each node is assigned at most one value: $(\neg 0_i \vee \neg 1_i) \wedge (\neg 0_i \vee \neg 2_i) \wedge \ldots \wedge (\neg (k-1)_i \vee \neg k_i)$ for each i.
- Nodes connected by an edge must not both belong to $\{1, \ldots, k\}$: $(\neg 1_i \vee \neg 2_j) \wedge (\neg 1_i \vee \neg 3_j) \wedge \ldots \wedge (\neg (k-1)_i \vee \neg k_j)$ for each edge (i, j).
- Values 1 to k must appear at least once: $(1_1 \vee 1_2 \ldots \vee 1_n) \wedge \ldots \wedge (k_1 \vee k_2 \ldots \vee k_n)$
- Values 1 to k must not appear more than once: $(\neg 1_1 \vee \neg 1_2) \wedge (\neg 1_1 \vee \neg 1_3) \wedge \ldots \wedge (\neg 1_{n-1} \vee \neg 1_n) \wedge \ldots \wedge (\neg k_1 \vee \neg k_2) \wedge (\neg k_1 \vee \neg k_3) \wedge \ldots \wedge (\neg k_{n-1} \vee \neg k_n)$

2.1.3 The right level of details

Abstracting out too much can cause loss of details. If utilized, the details may result in a solution with better performance. At other times, the solution may be the same but the details may help make the problem with the same solution easier to solve. Selecting the right level of detail to abstract is often crucial. This can be difficult and as Shannon mentioned, over-abstracting and adding refinements to the solution is often an easier way to understand and attack a problem.

Using the right amount and type of information is often a significant research advance. For example, using the right type of features has made face detection practical for machine vision [16]. As another example, consider the model for biological sequence comparison described in Section 2.1.1. This model is usually called a global alignment model as it considers the best match for the entire sequence. As large part of the sequence are often irrelevant, a better model is a local alignment problem that asks for the best scoring match of a subsequence within each subsequence. This refinement in the formulation only requires a slight change in the algorithm but is considered a significant advance [14].

We will give an example where exploiting additional information allows us to construct a more efficient algorithm for solving the same problem.

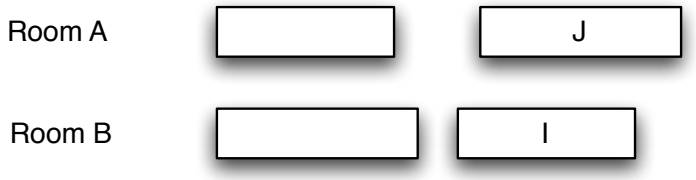
Classroom Scheduling: The School of Computing has n identical teaching rooms. Given the class and tutorial timetable for the semester, it is desirable to use the fewest rooms possible so that the remaining rooms can be designated as student rooms for students to use in between their classes. How do you minimize the number of tutorial rooms used? Assume that classes and tutorials are the same for the purpose of this problem.

Let's take the approach we have been using and reduce the problem into a graph colouring problem. 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 as follows:

- 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*[8] 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). This 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 changed an NP-complete one into a problem that can be solved using sorting.

So, how much detail is required for practical problems? That is, of course, problem dependent. But you will find that for most practical problems, the gains from adding more details become small fairly quickly. So, after capturing the most important details, a fair bit of hard work is required for additional small improvements and you may want to consider whether it is worth the effort (insights that lead to large improvements are then particularly valuable).

2.2 Principle of Generality

Let's say someone in industry asks you to solve a problem. In this case, this may be a problem that will make a difference to the the company - in some sense, this is a real problem that matters

[8]Considering extreme cases, such as the *first*, *last*, *largest*, *smallest*, etc., is often a useful problem solving technique.

to someone, rather than an academic one. You may be tempted to solve just that problem, make the person happy and be done with it. However, if you are looking to make research impact, 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. This is one way to make a problem that you have o solve more useful and important.

Paradoxically, making a problem more general sometimes makes it easier for a human to solve. This is often because the more general problem has less details that can distract the human from solving the problem. For example, in the protein similarity problem, the 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 is relatively easy to solve, even by hand.

2.3 Example: Google PageRank and Random Walk on Graphs

How do you rank web pages on the WWW for search? It was proposed in [9, 1] that the link structure of web pages can be exploited to improve ranking. A link from one page to another can be seen as conferring importance to the page being linked to. Viewed this way, the WWW is just a large graph with web pages as vertices and links as directed unweighted edges.

What is a reasonable criterion for ranking the vertices within such a graph? Page et. al. [9, 1] proposed that we can use the probability of finding a surfer on the web page as a measure of its importance. To do that, you need a model of how web surfers go to web pages. They proposed a very simple model where the surfer move from one page to another page at each time instance. To select the page to move to, the surfer would randomly pick one of the links of the page they are currently on with equal probability and follow it to the next page.

This can be immediately recognized as a Markov chain, which is well studied. The Markov chain can also be represented as a Markov matrix, which is nonnegative with each column adding to 1. In a Markov matrix, the i-th entry of column j represents the probability of a surfer at vertex j moving to vertex i at the next time instance (summing over all vertices that the surfer can move to gives a total probability of 1). A Markov matrix A is known to have the following properties [15]:

1. $\lambda_1 = 1$ is an eigenvalue
2. its eigenvector x_1 is nonnegative - and it is a steady state since $Ax_1 = x_1$
3. the other eigenvalues satisfy $|\lambda_i| \leq 1$
4. if any power of A has all *positive* entries, these other $|\lambda_i|$ are below 1. For any initial probability vector u_0, the solution $A^k u_0$ approaches a multiple of x_1 - which is the steady state u_∞.

The random surfer model described above is oversimplified when used as a measure of web page importance. Web pages with no outgoing links acting as sinks that will trap all the surfers. Furthermore, surfers often go to other pages instead of following a link on the current web page.

Additional details are easily added to fix these issues (see [7, 9, 1] for details). The details were also selected so that the resulting model satisfies the last properties in the list above, immediately giving a practical iterative algorithm for computing the probability.

2.4 Exercise

1. Formulate Sudoku as one of the graph problems described in the notes.

2. Technopreneurs are making a lot of money from the internet these days. You would like to join in on the gold rush; so you are always looking out for good ideas. Your friends complain that their internet access to many popular sites are very slow. This gives you an idea! Why not set up a service to allow companies to place copies of their web sites at many locations all over the world? This way the companies can ensure that their web pages are delivered to any location in the world quickly. You decided that a 400ms average round trip time is acceptable to most users. Now you need to find out where to place your servers with copies of the web pages such that the average round trip from any city to the nearest server is no more than 400ms.

   - Model this problem as a graph problem described in the notes.
   - Suggest possible improvements to the model by adding in some of the information abstracted away. Do you think these improvements would be practically significant?

3. In DNA sequencing, we obtain many contiguous fragments of DNA that needs to be re-assembled to obtain the complete DNA sequence. Different segments that come from the same section of the DNA sequence have overlaps between them, e.g. we may have fragments $aagtt$ and $ttcag$ with overlap tt of length 2. Assume that every part of the complete sequence is covered by some fragment and no fragment is a proper substring of another fragment (can be tested and removed). A naive formulation for finding the complete sequence is to find the shortest superstring containing all the fragments.

   - Reduce the shortest superstring problem to one of the graph problems in this note.
   - The formulation is not realistic in several ways. Can you suggest important details that have been abstracted away?

4. Certain word processors will automatically correct a wrongly typed word with a word that is similar to it from the dictionary. Use the analogy to the distance between protein problems to formulate a distance measure between a dictionary word and a typed word.

5. Discuss how exam scheduling can be formulated as a graph colouring problem assuming that there is no limit to the number of rooms that can be used at any time and that the aim is to minimize the exam period. Compare this formulation with the solution adopted by NUS.

6. Prof KnowItAll is proposing a new lossy image compression algorithm. An image is treated as an array of pixels. The array of pixels is partitioned into sub-arrays of length n. Each

sub-array is treated as coefficients of an n-dimensional vector in Euclidean space. Without compression, each vector would take $8n$ bits to transmit as each pixel can take 256 values (for simplicity only gray-scale images are considered). For compression, Prof KnowItAll proposes that a dictionary be constructed. The dictionary will be included in compression and de-compression programs. Instead of transmitting a vector, the index (position in the dictionary) of its closest approximation in the dictionary in terms of Euclidean distance will be transmitted instead. So, for a dictionary of size N, only $\log_2 N$ bits instead of $8n$ bits need to be transmitted for each vector. When de-compressing, the index that is received can be used to look up the appropriate dictionary entry to recover the transmitted vector.

To construct the dictionary, Prof KnowItAll proposes that a large dataset of vectors be collected. A subset of vectors in the dataset is then selected as the dictionary entries under the condition that every vector in the dataset has a Euclidean distance of no more than d from its closest approximation in the dictionary. The smallest such dictionary is desired in order to minimize the index length.

- Model this problem as a graph problem described in the notes.
- Suggest possible improvements to the model by adding in some of the information abstracted away. Do you think these improvements would be practically significant?

7. Encode the n-Queens problem as a SAT problem. Use your knowledge that each row must have exactly one queen to obtain a simpler encoding compared to the independent set encoding in the notes.

8. CNF can be used to build any formula, including those for adders and comparators. Discuss how this may be used to encode the independent set problem.

9. SAT is a decision problem where the answer is *true* or *false*. We are often interested in minimizing, e.g. in the optimal colouring problem, or maximizing, e.g. in the maximum independent set problem. Suggest a way for using SAT for these problems.

3 Formulation for Systems

When you are building a system rather than solving a problem, you are creating a tool which can be used for solving problems[9]. As you can imagine, formulating a systems problem can often be very difficult. You are trying to create a tool. Many considerations are involved, including correctness, performance/scaling, reuse, reliability, and ease of use. We will discuss some of the considerations that you should take into account when conceptualizing such systems.

[9]Naturally, there is no clear cut boundary for what is a system and what is an application. To build a system you may build second system that provides you with the tools to simplify the task of building the first system; in this case the first system is an application to the second system.

3.1 Principle of Modularity

The main idea is to break large systems into smaller parts so that each part is easier to build. If each part is small enough, it can be tested thoroughly to ensure correctness.

This is a very common way for building systems. For example, computer systems are often designed using a layered architecture.

1. Resistors, capacitors, transistors, used to build gates
2. Logic gates used to build control and data paths of processor
3. Machine language using binary representation
4. Assembly language
5. Programming languages
6. Applications

An example of a real operating system that is designed in layers is the *THE (Technische Hogeschool Eindhoven)* system [3]. It is an influential early multiprogramming system designed by a team led by Edsger Dijkstra. *THE* 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.

Dijkstra claimed that by making sure that the lower layer is correct before a higher layer is built, he is able to ensure correctness of the large system. As another example, 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.)

3.2 Information Hiding Principle

There are a lot of advantages in using a modular design. We do this all the time in our everyday in other aspects of our lifes, so this seems obvious. However, there is a non-obvious part. What is the criteria to be used for decomposing the problems? The currently accepted criteria for decomposing problems, advocated by Parnas [10] is called *information hiding*.

Parnas: ..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.

This insight is not obvious and is hard won through the experience of many. For example, 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 is one of the main reasons computer systems have been able to improve so quickly. By having an abstract layer between hardware and software, it is possible to take advantage of the exponential improvement in hardware without having to redevelop the software each time the hardware improves.

3.3 End-to-End Principle

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 by using 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? First of all, 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. So, the effect of the other steps is to decrease the frequency of retries, not eliminate the need for it. In fact, 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.

The principle of modularity tells us to decompose a large task into smaller tasks and hide the information within each task using abstract interfaces. The question that the end-to-end principle [11] 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.

The use of the principle is clearest in communication systems for applications such as error recovery and security using encryption. However, it is equally applicable in many other system design problems. For example, 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 argument here is that the computer designer is unlikely to be able to predict the instructions that will be most useful for any application and optimizing the instruction set for one application is likely to lead to poorer performance on other applications. So, it is better to provide the simple but fast instructions and leave the construction of complex functionalities from simple instructions to the application which has the information to optimize better.

A related principle is the principle of least power when choosing a representation. This principle suggests choosing the least powerful representation system when selecting a representation. The less powerful the representation, the easier it is for other people or programs to manipulate the data. Tim Berners-Lee selected HTML as the representation language for the web because it is easy for different programs to do different things with it, including presentation, indexing, extracting tables and other tasks[10].

3.4 Principle of Simplicity

Whatever problem we are formulating, it is useful to keep the principle of simplicity in mind. When there is any doubt, always go for the simplest thing. Simpler things are easier to understand and

[10] `http://www.w3.org/DesignIssues/Principles.html`

less likely to contain bugs when implemented. Complex things take longer to design, implement, test and debug. Performance tuning is also more difficult as you cannot improve what you do not understand.

This is also known as the KISS principle: Keep It Simple, Stupid! But, of course, there is a limit to simplicity - your system should be complex enough to perform the task it is required to do.

Simplicity may be relative. Something can be simple relative to what is known or expected - it does not convey or require new information. This variant, 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.

3.5 Example: Relational Database

We will now look at relational database as an abstraction for a database system from the viewpoint of the principles mentioned above.

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.

An example of a relation is shown in Table 2.

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

Table 2: Example of a relation.

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 [2]. For example, let us compare it with a hierachical (tree) representation. For example, in Table 2, there are nine possible trees for representing this data: three each rooted by *Student*, *Lecturer* or *Course*. Imagine a database using one of the tree representations where the representation is exposed to data accessing programs. 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.

Relational algebra, consisting of operations such as selection, projection, join, union, difference and renaming gives a query language for the relational model. A nice property of this model is that the output of an operation on a relation is also a relation (closed under the operations). This makes the relational model simple and elegant. However, 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. Should we abandon relational algebra as a query language? The end-to-end principle suggests that functions that are rarely used can be implemented at a higher level to keep the abstraction simpler. This suggests using relational algebra augmented with some commonly used operators as the query language, and leaving some of the more complex queries for the consumer of the data to sort out. Indeed, earlier version of query languages such as SQL implements a superset of relational algebra but did not allow queries such as transitive closure.

An abstraction should be as simple as possible but must be able to do the most important functions. There is one important characteristic of databases cannot be ignored - data is modified, inserted and deleted all the time. To help handle these 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. The success of relational databases shows that the relational model is roughly the right abstraction for most database problems in the last 30 years.

3.6 Exercise

1. The Java virtual machine adds a layer of abstraction between Java programs and the underlying computer system. Discuss the advantages and disadvantages.

2. The *model-view-controller* design pattern is often used in user interface design. In this pattern, the model contains the data structure and logic for the application - quite often these are databases. The view renders objects into visible forms and the controller processes user requests and pass them to the model. Relate this design to the information hiding principle.

3. One of X Window system's principle of design is to provide "mechanism, not policy". For example, it provides the mechanism for manipulating a window but does not say how to do the manipulation (policy). As a result it is very easy to change the looks in X Window. But it also resulted in many applications that worked differently and did not work well together (different policies). Comment on this from the view of the information hiding principle and principle of least surprise.

4. Unix programs are often written to accept a text stream as input and output a text stream. This allow various programs to be piped together easily, e.g. `ls | grep CS2306 | sort -r` will do a directory listing, select the lines that contains "CS2306" and sort the lines in reverse order. Discuss the use of text as input and output in terms of the information hiding principle and principle of least power.

5. Discuss XML and relational databases, viewed from the point of views of the information hiding principle and the end-to-end principle.

6. Discuss the following quote from Mike Haertel: *Any performance problem can be solved by removing a level of indirection.*

4 Formulation for Theory

Theory is essentially mathematical analysis of problems. It is used both in systems as well as applications. When formulating a theoretical question, you should look to provide insights - understanding of the phenomenon that you are looking at.

For theory, the principle of simplicity applies very strongly. It is usually only possible to prove results when the formulation is simple enough. Sometimes simplicity is prefered over realism as long as the insights obtained from the simplified problem is useful for understanding the phenomenon that is being studied.

4.1 Worst Case Principle

This is probably the most commonly used simplifying assumption. There are often too many cases to analyse. Analysing all of them often does not provide useful understanding as it is unclear which of the cases occurs in practice. In such cases, worst case analysis often provides useful insights.

It is occasionally interesting to look at the best case outcome. However, the worst case is often more useful as it provides us with a guarantee regardless of what happens.

You should already have some familiarity with worst case analysis of algorithms. For example, insertion sort has best case runtime of $\Theta(n)$ but worst case runtime of $\Theta(n^2)$. Merge sort has a worst case runtime of $\Theta(n)$ - this is often used to justify prefering merge sort over insertion sort.

4.2 Asymptotic Principle

Things often becomes simpler when the number of elements becomes large. One way this happens is because, when we look at what happens when thing become large, only the rate of growth matters. For example, in analyzing algorithms, we normally ignore constants in the running time. For example, insertion sort has worst cases runtime bounded by $K_1 n^2$ and merge sort has worst case runtime bounded by $K_2 n \log n$ where K_1 is probably smaller than K_2. However, we normally ignore the constants K_1 and K_2 as the function n^2 grows much faster than $n \log n$.

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 $\Omega(f(n))$.

4.3 Comparative Principle

Quite often, the absolute value of something does not provide useful insights but the value in comparison to something reasonable tells us more.

For example, in computer systems, there is often a fast memory (e.g. main memory) which is small and slow memory (e.g. disk) which is large. To allow large problems to be solved, it is often assumed that memory is as large as the slow memory. Fast memory is used as temporary storage to store commonly used items to speed up computation. When the processor needs an item, it checks in fast memory first. If the item is present in fast memory, the processor can continue processing, otherwise it has to fetch the item from slow memory. How do we decide on the set of items to keep in fast memory? A commonly used method is to replace the least recently used (LRU) item with the item just fetched from slow memory. Another is to replace the least frequently used (LFU) item. How do we compare the algorithms? It is difficult to characterize the distribution of items that will be fetched, so probabilistic analysis is quite difficult. A worst case deterministic analysis will indicate that all algorithms are the same because in the worst case all items fetched can be different.

It turns out that it is possible to do *competitive analysis* for these types of problems, where an algorithm is competitive if the cost of the algorithm is within a constant times the cost of the best possible algorithm [13]. Performing analysis relative to the best algorithm makes sense in this case: if the best algorithm performs poorly, then our algorithm is also allowed to perform poorly. Under such a framework, it turns out that LRU is competitive whereas LFU is not.

5 Example: Theory of NP-Completeness

The most famous, and arguably the most important problem in theoretical computer science, is the question of whether $P = NP$? The class P refers to problems that can be solved in polynomial time. The class NP refers to problems whose solution can be verified in polynomial time. The question of whether $P = NP$ is roughly whether problems whose solution can be verified in polynomial time can also be solved in polynomial time. Most computer scientists believe that $P \neq NP$ [5]. A few believe that it is possible that $P = NP$. No one currently have a good idea on how to prove it either way.

Let's look at how the the problem is related to some of the principles we have discussed. First, the question is formulated as a worst case problem: In the worst case is there any problem in NP that cannot be solved in polynomial time? For practical problems, it is unclear if the worst case formulation is the most useful formulation - nevertheless other formulations such as average case formulations are often difficult to do. Furthermore, worst case analysis provides useful insights as we are often interested in having worst case guarantees.

Second, the question is formulated asymptotically. The question is not about a fixed sized problem, but about how the the computation time grows as the problem size grows. The asymptotic formulation makes the entire theory simpler and also makes proving results within the theory simpler.

Third, one of the most useful aspect of the theory is that it allows us to claim that a problem

is difficult, even though we do not know its actual worst case runtime. This is because we have techniques that allow us to claim that a problem is as difficult as the hardest problem in NP (we will go into the technique later in the course). This illustrates the use of the comparative principle to make claims that are insightful even when we cannot make useful claims about the absolute value of the quantity we are interested in.

References

[1] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems*, 30(1-7):107–117, 1998.

[2] E.F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):387, 1970.

[3] E.W. Dijkstra. The structure of the THE-multiprogramming system. In *Proceedings of the first ACM symposium on Operating System Principles*, pages 10–6. ACM, 1967.

[4] R.P. Feynman, R. Leighton, and E. Hutchings. " Surely you're joking, Mr. Feynman!". 1985.

[5] L. Fortnow. The status of the P versus NP problem. *Communications of the ACM*, 52(9):78–86, 2009.

[6] B.W. Lampson. Hints for computer system design. *ACM SIGOPS Operating Systems Review*, 17(5):33–48, 1983.

[7] C.D. Manning, P. Raghavan, and H. Schutze. *Introduction to information retrieval*. Cambridge University Press Cambridge, UK, 2008.

[8] S.B. Needleman and C.D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of molecular biology*, 48(3):443–453, 1970.

[9] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. 1999.

[10] D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1058, 1972.

[11] J.H. Saltzer, D.P. Reed, and D.D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems (TOCS)*, 2(4):288, 1984.

[12] Claude E. Shannon. Creative thinking. *Mathematical Science Center, AT&T*, 1993.

[13] D.D. Sleator and R.E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):208, 1985.

[14] T.F. Smith and M.S. Waterman. Identification of common molecular subsequences. *J. Mol. Bwl*, 147:195–197, 1981.

[15] G. Strang. Linear algebra and its applications, 1988. *Hartcourt Brace Jovanovich College Publishers*.

[16] P. Viola and M. Jones. Rapid object detection using a boosted cascade of simple features. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, volume 1, 2001.

Appendix: Abstract things are hard to think about ☺

One disadvantage of abstract representations is that they can be hard to think about because we are not familiar with them. It is sometimes helpful to map the abstract representation into things that are familiar to you to allow you to think more easily about them. Here's a little story by Richard Feynman [4]:

> *Topology was not at all obvious to the mathematicians. There were all kinds of weird possibilities that were 'counterintuitive.' Then I got an idea. I challenged them: 'I bet there isn't a single theorem that you can tell me – what the assumptions are and what the theorem is in terms I can understand – where I can't tell you right away whether it's true or false.'*
>
> *It often went like this: They would explain to me, 'You've got an orange, OK? Now you cut the orange into a finite number of pieces, put it back together, and it's as big as the sun. True or false?'*
>
> *'No holes?'*
>
> *'No holes.'*
>
> *'Impossible! There ain't no such a thing.'*
>
> *'Ha! We got him! Everybody gather around! It's So-and-so's theorem of immeasurable measure!'*
>
> *Just when they think they've got me, I remind them, 'But you said an orange! You can't cut the orange peel any thinner than the atoms.'*
>
> *'But we have the condition of continuity: We can keep on cutting!'*
>
> *'No, you said an orange, so I assumed that you meant a real orange.'*
>
> *So I always won. If I guessed it right, great. If I guessed it wrong, there was always something I could find in their simplification that they left out.*
>
> *Actually, there was a certain amount of genuine quality to my guesses. I had a scheme, which I still use today when somebody is explaining something that I'm trying to understand: I keep making up examples. For instance, the mathematicians would come in with a terrific theorem, and they're all excited. As they're telling me the conditions*

of the theorem, I construct something which fits all the conditions. You know, you have a set (one ball) – disjoint (two balls). Then the balls turn colors, grow hairs, or whatever, in my head as they put more conditions on. Finally they state the theorem, which is some dumb thing about the ball which isn't true for my hairy green ball thing, so I say, 'False!'

If it's true, they get all excited, and I let them go on for a while. Then I point out my counterexample.

'Oh. We forgot to tell you that it's Class 2 Hausdorff homomorphic.'

'Well, then,' I say, 'It's trivial! It's trivial!' By that time I know which way it goes, even though I don't know what Hausdorff homomorphic means.

I guessed right most of the time because although the mathematicians thought their topology theorems were counterintuitive, they weren't really as difficult as they looked. You can get used to the funny properties of this ultra-fine cutting business and do a pretty good job of guessing how it will come out.