Solution

After formulation, we would normally proceed to try and obtain some sort of useful solution. For systems, this may be an implementation of the system. This is often an engineering problem, with many ways to obtain a solution.

For applications, we often try to solve a whole class of problems, instead of just one problem, so the solution is often a method or an algorithm. For example, we are interested in obtaining a sorting algorithm, rather than just a sorted list. An algorithm is simply a method for solving a class of problems quickly. It often exploits the same types of properties that humans apply when solving problems by hand. Hence, the ideas discussed here are often also useful for solving problems by hand, for example for doing proofs.

1 System Problems

1.1 Implementation Principles

The following list of rules by Rob Pike is taken from [5]:

Rule 1: You can't tell where a program is going to spend its time. Bottlenecks occur in surprising places, so don't try to second guess and put in a speed hack until you've proven that's where the bottleneck is.

Rule 2: Measure. Don't tune for speed until you've measured, and even then don't unless one part of the code overwhelms the rest.

Rule 3: Fancy algorithms are slow when n is small, and n is usually small. Fancy algorithms have big constants. Until you know that n is frequently going to be big, don't get fancy. (Even if n does get big, use Rule 2 first.) For example, binary trees are always faster than splay trees for workaday problems.

Rule 4: Fancy algorithms are buggier than simple ones, and they're much harder to implement. Use simple algorithms as well as simple data structures.

The following data structures are a complete list for almost all practical programs:

- array
- linked list
- hash table
- binary tree

Of course, you must also be prepared to collect these into compound data structures. For instance, a symbol table might be implemented as a hash table containing linked lists of arrays of characters.

Rule 5: Data dominates. If you've chosen the right data structures and organized things well, the algorithms will almost always be selfevident. Data structures, not algorithms, are central to programming. (See Brooks p. 102.)

Rule 6: There is no Rule 6.

On Rule 4, Ken Thompson says: *When in doubt, use brute force.*

1.2 Plan to Throw One Away

You will throw one away anyway. Particular on research projects, where there are a lot of innovations. Having a prototype can be a lot cheaper. Later systems usually are able to reuse a lot, especially ideas, from earlier systems.

1.3 Back of the Envelope Calculations

It is important to be able to do a rough estimate of the performance of a system without having to build it. Some useful numbers and an example [3]:

- L1 cache reference: 0.5ns
- Branch mispredict: 5ns
- L2 cache reference: 7ns
- Mutex lock/unlock: 100ns
- Main memory reference: 100ns
- Compress 1K bytes with Zippy: 10,000ns
- Send 2K bytes over 1Gbps network: 20,000ns
- Read 1MB sequentially from memory: 250,000ns
- Round trip within the same datacenter: 500,000ns
- Disk seek: 10,000,000ns
- Read 1MB sequentially from network: 10,000,000ns
- Read 1MB sequentially from disk: 30,000,000ns
- Send packet California $\rightarrow$ Netherlands $\rightarrow$ California: 150,000,000ns

Evaluate two designs for generating 30 image thumbnails from disk:

Design 1: Read serially, thumbnail 256K images

```
30 seeks * 10 ms/seek + 30 * 256K / 30MB/s = 560ms
```

Design 2: Read in parallel

```
10 ms/seek + 256K / 30 MB/s = 18 ms
```

1.4 System Efficiency Tricks

The following are from [4]:

Handle normal and worst case separately: You should make sure that the normal cases are fast. Worst cases are usually rare, and as long as it works, the outcome is often acceptable. Handling them separately is advisable as they are usually quite different and trying to do them together would usually penalize the usual case.

Use dedicated resources: Dedicating resources rather than sharing them, when posible, is often a good way to improve efficiency. The addition of the graphics card means that the CPU is free from performing the graphics computation and can perform the computation of other parts of the program. By using a disk controller, the CPU is freed from having to manage the reading of data from disks. This is essentially a use of parallelism. Also of divide and conquer - dividing the tasks and handling them to different processors to use.

Cache answers: The idea is to store expensive computation and reuse it when required to save the recomputation time. An issue that often arise is that there is often not enough memory to store the result of all the computation. In that case, locality is exploited to store items that are likely to be reused, while those that are less likely to be used are discarded and recomputed when required.

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. This assumes temporal locality i.e. that recently used items (similar with respect to time of access) are likely to be used again. When fetching memory from slow memory, a whole block of memory is fetched, instead of only the required item. This assumes spatial locality i.e. items that are physically nearby in memory are likely to be accessed together.

Compute in background and use batch processing: For interactive systems, it is best to compute as little as possible before responding. When possible, work should be deferred to the background. For example, emails are often delivered and fetched in the background. Many

bank systems will take your requests for fund transfer and process it in the background later. This allows things to be done in batch, which is often much more efficient.

Safety first: It is very difficult to optimize the use of resources for general purpose systems as we cannot predict loads very well. A system usually cannot perform well if load exceed two-third of the capacity. Usually, it is better to design to avoid disaster, than to try to optimize, particularly when the cost of hardware is cheap and getting cheaper.

For example, in the first paging systems, memory was expensive and designers tried to optimize by clever swapping: putting related procedures on the same pages, predicting the next page to be referenced, running jobs with the same data together, etc. This was never very successful - the only really important thing was to avoid thrashing (pages being swapped in and out all the time due to insufficient memory).

Shed load: It is better to shed load to control demand than to allow the system to become overloaded. When a system is overloaded, nothing works. If load is shed, at least some of the processes can work and those that are refused service can try again later and do productive things in the mean time, rather than wait for service. Examples of this is for interactive system to refuse new users, or networks to discard packets.

2 Application Problems

Application problems are often formulated as search problems, e.g. find a solution that optimizes some criterion for an abstract problem.

Before looking at search methods, we consider what makes a useful representation for search.

2.1 State Representation

At any time in the computation, we need to represent what has happened previously. This gives rise to the notion of a state - one of the most useful notions throughout computer science.

Definition 1 *A state is a representation of a problem that summarizes all the computation done in the past so that future computation depends on only the state and not the history of computation done up to the current time.*

It is often useful to keep the notion of a state in mind when deciding how to represent your problem for a computer to solve. We first give some examples of different representions of states in different problems.

Example 1 In the game of chess, the current board position is the state of the game. The game path that led to the current position does not affect how the game is played in future.

Example 2 In the 8-queens problem, the position of the queens already placed on the board can be a state. Note that the order in which the queens were placed is irrelevant to future computation of finding a legal placement and is not stored.

Example 3 In the shortest path problem of driving from one town to another, the town that you are current at can be as state as how you get there is irrelevant to how you can get to the destination.

In doing search, we often define a *state space* and search over the space.
The notion of state is useful for many problems, not just when doing search.

Example 4 When an interrupt occurs in a CPU, the content of important registers are stored in memory before the interrupt is serviced. These form the state of the process and is restored when the current process is continued.

Example 5 HTTP is a stateless protocol because each request is independent of the request before or after it. However, many applications requires the state to be kept so that proper action can be taken given what has happened previously. This is implemented by storing the state in cookies in the client and transmitting the cookies to the server.

2.2 Divide and Conquer

Consider the 8-queens problem. Assume that we decided to represent the state as the partially completed board position with 8 or fewer queens placed on the board. How should we search this state space? The *divide and conquer* principle says that we should split up the problem into pieces of roughly equal size and search each piece for the solution. For simplicity, assume that we decided to search the space by putting the first queen in the first row, followed by the second queen in the second row, etc. One way to split the problem up into 8 roughly equal pieces would be to use each possible position of the queen in the first row: each piece of the problem here is the problem of placing the remaining queens on the board to solve the problem. Figure 1 shows the split for the 4-queens problem.

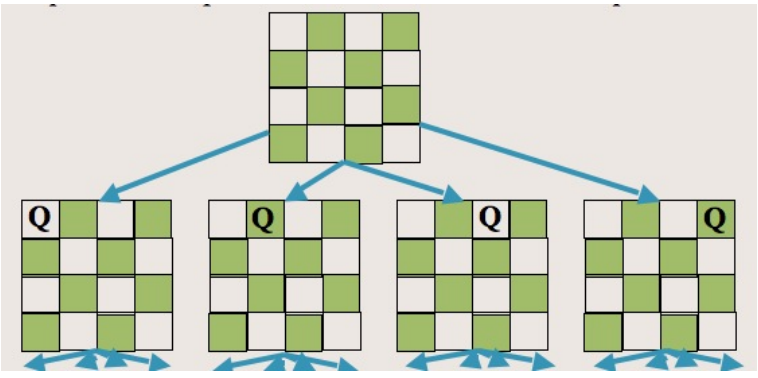

Figure 1: One way to split the state space into four groups.

Doing the search recursively is relatively easy. Algorithm 1 gives simple pseudo-code to do the search: it checks whether the current board can be completed to a legal queen placing with the addition of one queen. It assumes that a routine IsLegal is available to check whether a board contains legal placement of all the queens. To check whether a solution exists using ContainsQueenSol,

we simply need to check whether adding a queen at each column of the first row from an empty board can lead to a legal solution. Note that the current space of possible solution to search is divided roughly equally at every step of the recursive process.

Algorithm 1 CONTAINSQUEENSOL($board, row, column$)

```
Place queen at position (row, column) on board
if row = n then
  return ISLEGAL(board)
else
  for i = 1 to n do
    if CONTAINSQUEENSOL(board, row + 1, i) then
      return true
  return false
```

Note that the entire state space is often not searched by the program: it terminates as soon as a solution is found. We can do better: if we check whether the current partially filled board already fail some constraints (e.g. two queens already attacking one another), we do not need recursively compute its children. This often called *pruning* and is shown in Figure 2 where none of the child of node c can contain a solution. This type of search is often called *backtracking* search because the search goes back to an earlier state and tries again after meeting with a failure.

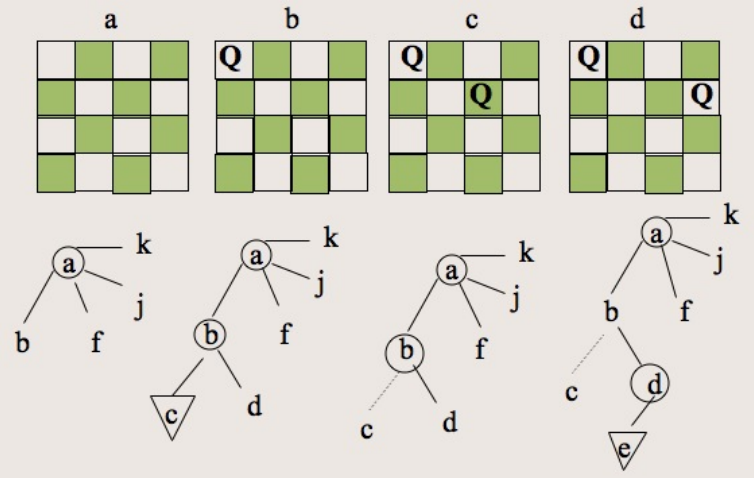

Figure 2: None of the children of node c can be solution.

In this example the search continues and finally finds a solution as shown in node i of Figure 3.

2.3 Most Constrained Variable

In the n-queens example, we use placed queens row by row from the top to bottom row. We are actually free to pick a different row to place the next queen at anytime. Which row should we pick

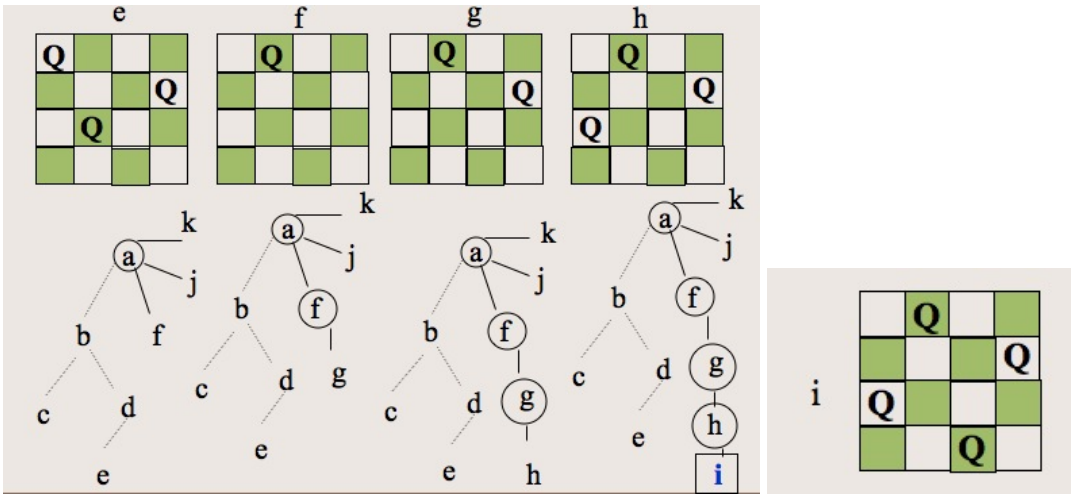

Figure 3: Fail at node e and finally success at i.

at any time? The *most constrained variable principle* says that we should pick the most constrained variable to assign value to at any state. The rationale behind the principle is that if we need to fail, we want to fail early but if we are successful with assigning the correct value, the remaining search space becomes much smaller.

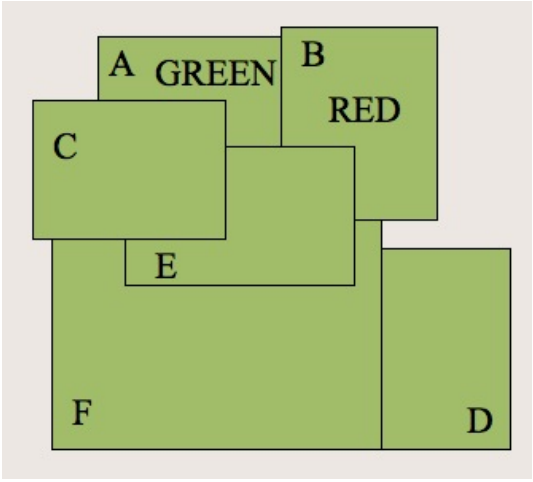

Figure 4: Partially coloured map.

Consider the map colouring problem in Figure 4 where countries that have common boundaries cannot be coloured the same. Assume that we are trying to colour the map using three colours. The variables in this case are A, B, C, D, E, and F. To select the variable that is most constrained, define a function $f(s, v)$, where s is the state and v is the variable, to count the number of remaining option to color the variable without violating the current constraints in the current state. For example: $f(s, E) = 1$, $f(s, C) = 2$, $f(s, F) = 2$, $f(s, D) = 3$ in Figure 4. The most constrained

variable is E. Colouring *E BLUE* and further using the principle leads to *C RED* and *F GREEN* and *D RED* without needing to backtrack in this case.

Going back to the n-queens example, the variables are the rows $r_1, \ldots, r_n$ and each variable r_i can take n possible values, representing the column of the queen at row i. By the most constrained variable principle, one way to select the row r_i to process next is to select the row with the least number of possible locations to place the queen without violating the constraints.

2.3.1 Example: Satisfiability

The satisfiability (SAT) problem asks whether there an assignment of variables that will make the sentence true? One of the common ways of solving satisfiability is to search for a satisfying assignment, similar to what we have just discussed. We often represent the boolean formula in CNF form e.g. $(x \vee \neg y) \wedge (y \vee z) \wedge (\neg x)$. One very successful algorithm is the Davis-Putnam algorithm. The state is the current partial assignment of variables. The algorithm branches on the values of each variable $x_1, \ldots, x_n$.

In SAT, we can check early to see whether a satisfying assignment exists: if every clause is already true even with a partial assignment of variable, then we can stop and return true. Similarly, if one clause evaluates to false, we can stop the current partial assignment and backtrack as it has already failed the constraints.

An application of the most constrained variable heuristic is the *unit clause heuristic*. A unit clause is a clause with one literal. In a unit clause, the variable is constrained to take the value to make the literal true, hence should be assigned first. On assignment, this can cause other clauses to become unit clauses and the forced assignment is continued - this process is called unit propagation and can result in considerable computational savings.

2.3.2 Example: Most constrained heuristic for human problem solving

Human problem solving often use the most constrained heuristics in various forms: looking at things that are *largest*, *smallest*, *first*, *last*, etc. For example, in the proof that the greedy algorithm for interval colouring is optimal, we looked at the *first* time the algorithm differs from the optimal solution. The *working backwards heuristic* for solving problems often works well because the end point is often more constraining than the starting point, reducing the search required.

2.4 Exploiting Bounds

Consider doing backtracking search to do a minimum dominating set problem. Assume that the state is a subset of selected vertices that can potentially be extended into a dominating set. We can prune off the search tree and backtrack if the current state contains more vertices than the best solution found in the search so far. Can we do better than using the best solution found so far for pruning?

If we remove the subset of vertices that has already been dominated by the subset of vertices selected in the current state, we will be left with a subgraph. Let the number of remaining vertices be N and let d be the largest number of vertices that can be dominated by a single vertex in

the subgraph. Then N/d gives an optimistic (smaller than required) estimate on the number of additional vertices required to dominate the original graph. If the sum of the current selected vertices and the estimate is more than the current best, then we can prune this part of the search tree as it cannot be better than the best solution found so far.

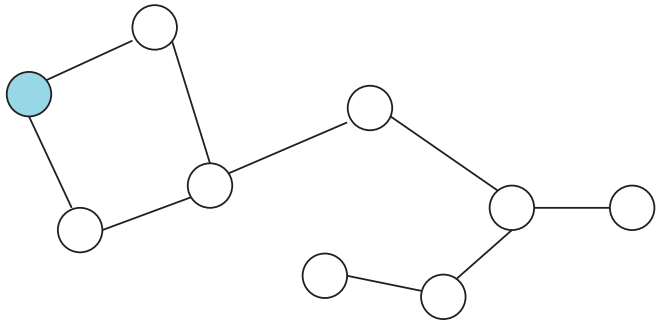

Figure 5: The shaded vertex dominates the two vertices connected to it. The remaining subgraph requires three additional vertices to dominate.

Figure 5 provides an example. In the figure, the shaded vertex has already been selected and dominates itself and the two vertices connected to it. The remaining six vertices requires three vertices as a dominating set. The most that any vertex can dominate is 4 vertices, suggesting an optimistic estimate of 2 vertices to dominate the remaining six vertices.

This technique is known as *branch and bound*. We divide the current set of potential solutions into subsets. For each subset, we provide an optimistic estimate. If the estimate is poorer than the best solution found so far, we can stop and backtrack. Otherwise, we recursively split the subset and continue.

Can we squeeze more out of the bound? How about using it to order the nodes to be searched so that the nodes that looks the best in terms of the bound is searched first? Pseudo-code for doing best first search is given in Algorithm 2.

Algorithm 2 BESTFIRSTSEARCH(G, S)

```
found = φ, queue = {S}
while queue ≠ φ do
  n = EXTRACTBEST(queue)
  if ISGOAL(n) then
    return COST(n)
  for c ∈ NEIGHBOUR(n, G) do
    UPDATECOST(c)
    if c ∉ found then
      INSERT(c, queue)
      INSERT(c, found)
```

This gives a version of the A^* algorithm. If EXTRACTBEST gives the best node according to an optimistic bound, then the algorithm will return an optimal solution. To see this, note that *queue* contains all leaves of the current search tree that can be reached from the start state. The set of configurations represented by these leaves forms a *partition* of the possible configurations. Each of these leaves are valued according to an optimistic estimate of the best possible completion. The algorithm terminates if a full goal configuration is reached and selected to have the best estimate. For a full configuration, the estimate is the true value since it is not a partial configuration - its true value is better than the optimistic estimate of all subsets that have yet to be explored, hence is optimal.

How does the branch and bound and A^* algorithms compare? A^* uses more information, so we expect it to be faster. What about the amount of memory required? For branch and bound, we only need to store the best solution so far, and the path from the root to the current state. A^* needs to store all the leaves in the current tree - this can be very large (usually exponential in the depth of the tree) and A^* often runs out of memory for larger problems.

A^* is often used to find a path from the start state to the goal state, instead of just finding a state satisfying some property. In this case, the estimated cost consist of two components $f(n) = g(n) + h(n)$ where $g(n)$ is the actual cost of the path to the current vertex and $h(n)$ is an estimate of the minimum cost of the path to the goal from n. If h is optimistic, it is called *admissible*. To be able to reconstruct the path, additional information needs to be kept, usually a pointer to the parent of the node on the path. Algorithm 2 also needs to be modified when the object being searched is a path instead of a state; instead of ignoring nodes in the list *found*, the cost of any node, including those in *found* need to be updated if the path to the node is improved by the latest path found. Alternatively, if the heuristic satisfies an additional condition, consistency or monotonicity, previously expanded nodes in *found* never need to be expanded again. A heuristic h is consistent if $h(n) \leq h(n') + cost(n, n')$ for every successor n' of n - this ensures that the optimal path to a node in *found* is always the one used when the node is placed in *found* [6]. Algorithm 2 works correctly with consistent heuristics.

Roughly, heuristic knowledge about the domain in the form of bounds can be used for improving search. By ensuring that the bound is optimistic, the search will be *complete*, i.e. the best solution will eventually be found. Being greedy with respect to an optimistic bound is often a good idea. However, as part of search, this can sometimes be expensive in terms of the space required.

2.5 Relaxation Principle

How do we construct optimistic heuristics? Optimistic heuristics can often be discovered by consulting simplified models of the problem domain. Relax the constraints to obtain simplified models - if the problem has fewer constraints, a solution with better value can often be found compared to a more constrained problem. A simple example is the straight line heuristic for estimating the distance to a goal in the road map problem. The straight line heuristic assumes that it is possible to move in a straight line to the goal while in the real model, motion is restricted to using the available roads.

The *eight puzzle* is shown in Figure 6. Two admissible and consistent heuristics are:

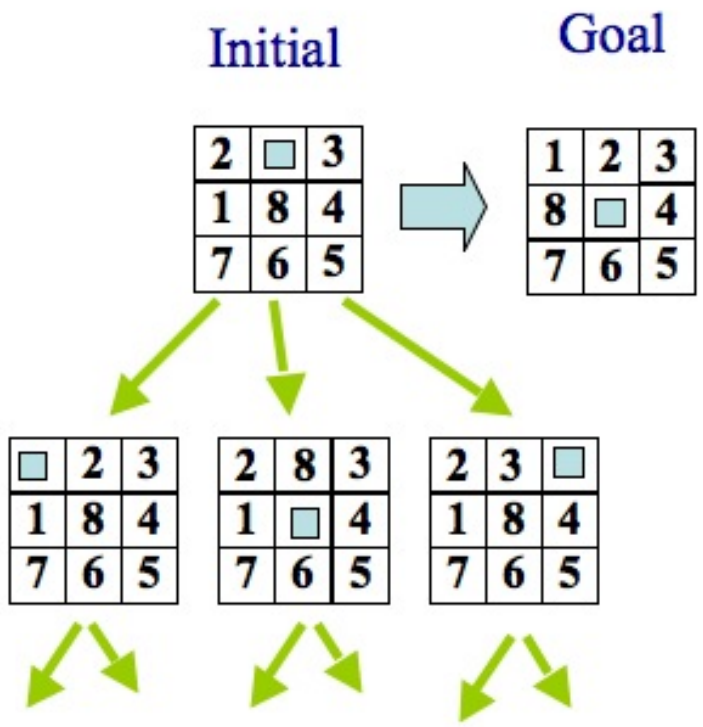

Figure 6: Eight puzzle, showing the initial and end states.

- h_1: Measure the number of tiles in the wrong position. Each step can only move one tile, so number of tile in wrong position is an optimistic estimate of number of moves. If we change the rule in the 8-puzzle such that the tile could move anywhere and not just to adjacent empty square, h_1 gives the number of moves required.
- h_2: Measure the sum of distances of tiles from their goal position. This is admissible because each move can only move one tile one step closer to its goal. If we can move one square in any direction even onto an occupied square then h_2 gives the number of steps required.

Both the heuristics can be thought of as solution to relaxed version of the original problem.

2.6 Greedy Algorithm

The A^* algorithm is an example of a greedy algorithm that eventually gives the optimal solution. A greedy algorithm usually makes the move that optimize the current cost without backtracking when it later does not give the optimal solution. The solution to the classroom allocation problem in an earlier lecture is a greedy algorithm that gives an optimal solution. However, greedy algorithm can fail to find the optimal solution. Consider the following problem:

Coin changing: Find the fewest number of coins required to obtain n cents. There are k denomination $d_1, \ldots, d_k$.

One possible greedy algorithm is:

- Find the largest denomination d_i smaller than or equal to n
- Set n to $n - d_i$ and repeat.

This greedy algorithm fails when the denomination is 1 cent, 10 cents and 25 cents and we need the fewest number of coins needed to obtain 30 cents. The greedy algorithm would pick $25 + 1 + 1 + 1 + 1 + 1$ while the optimal solution is $10 + 10 + 10$.

Greedy algorithms are also often used for continuous optimization. If we want to maximize the function $f(x)$ with respect to x, one way is to update the current value of x by moving in the direction of the gradient of f at x. This is the direction that increases f the most locally and the method is called gradient ascent. These algorithms may get stuck in local optimums, locations where every direction will locally decrease the value of the function before the value can be increased.

2.7 Memoization and Dynamic Programming

Consider again the divide and conquer method for solving a problem. The solution of a problem is defined in terms of the solution of the subproblems. Sometimes the same subproblem is used multiple times in the computation. A simple improvement is to store the solution of subproblems so that each subproblem only needs to be computed once. If the number of subproblems is small, the computation time immediately becomes small.

Consider the coin-changing problem. Let $M[i]$ be the minimum number of coins required to construct i cents and the denominations be $d_1, \ldots, d_k$. To find the smallest number of coins required to construct i cents, we find the smallest number of coins to construct $i - d_j$ cents for $j = 1, \ldots, k$ and add one more coin. In other words, $M[i]$ satisfies the following relationship:

$$M[i] = \begin{cases} 1 + \min_{1 \le j \le k}\{M[i - d_j]\} & i > 0 \\ 0 & i = 0 \\ \infty & i < 0 \end{cases}$$

This translate easily to pseudocode in Algorithm 3. Define an array $M[0, n]$. Initialize $M[0] = 0$ and the initialize the other elements to ∞.

What is the running time of the algorithm? At most there are n distinct subproblems for $i = 1, \ldots, n$. Note that the execution graph of the problem is acyclic i.e. to solve for $M[i]$, you only need to solve for problems strictly smaller than i. So $M[i] \neq \infty$ by the time NUMCOINS(i) is called again - each subproblem is solved only once. Each subproblem calls NUMCOINS at most k times, so the part of the code that test whether $i < 0$ or whether $M[i]$ has been solved before is run at most nk times. The remaining part of the code has a loop that loops for k iterations and is done only once for each subproblem. As there are at most n subproblems, the total running time is $O(nk)$. In comparison, the recursive algorithm that does not store the solution to subproblems can take time that is exponential in n.

Algorithm 3 NUMCOINS(i)

```
if i < 0 then
    return ∞
else if M[i] ≠ ∞ then
    return M[i]
else
    minCoin = ∞
    for j = 1 to k do
        if minCoin > NUMCOINS(i − d_j) + 1 then
            minCoin = NUMCOINS(i − d_j) + 1
    M[i] = minCoin
    return minCoin
```

The technique of storing solution of subproblems that have been solved before is called memoization[1]. Memoization typically refers to storing solution to subproblems when used with recursive algorithms. Typically, the program can also be written in an iterative method; it is usually called *dynamic programming* when written in an iterative manner. A dynamic programming solution is shown in Algorithm 4.

Algorithm 4 NUMCOINS(n)

```
Initialize array M of size n + 1 to ∞
Set M[0] = 0
for i = 1 to n do
    for j = 1 to k do
        if (i − d[j] ≥ 0) and (M[i − d[j]] + 1 < M[i]) then
            M[i] = M[i − d[j]] + 1
```

An example run of the dynamic programming algorithm is shown below with $n = 9$ and denominations: $d[1] = 3$, $d[2] = 5$, $d[3] = 7$.

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
$M[i]$	0	∞	∞	1	∞	1	2	1	2	3

In this case, the minimum number of coins required is $M[9] = 3$. We can also recover the path that led to $M[9] = 3$. We store another array σ and change the innermost loop to

```
if (i − d[j] ≥ 0) and (M[i − d[j]] + 1 < M[i]) then
    M[i] = M[i − d[j]] + 1
    σ[i] = i − d[j]
```

Running the code, we get

[1]Memoization comes from the word *memo*, which is a document typically used for communication in a company.

i	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
$M[i]$	0	∞	∞	1	∞	1	2	1	2	3
$\sigma[i]$	∞	∞	∞	0	∞	0	3	0	3	6

Starting from $\sigma[9]$ we see that the last element in the path is constructed using $M[6]$ and reading $\sigma[6]$ we see that it is constructed using $M[3]$ giving the path $M[3], M[6], M[9]$ which gives three 3 cent coins.

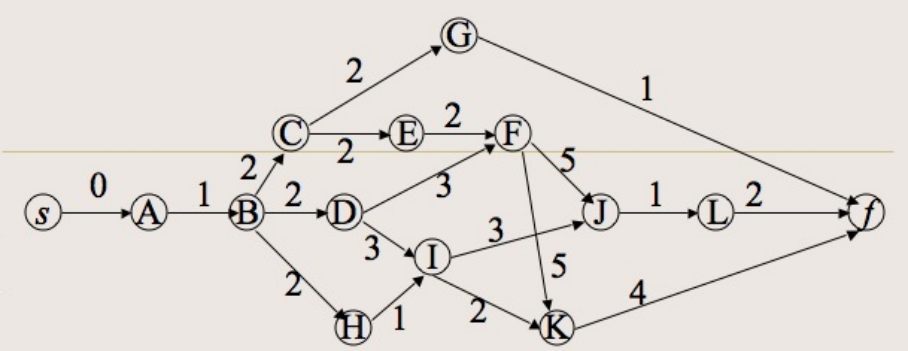

Figure 7: Longest path problem.

We now consider the longest path problem shown in Figure 7. Let's solve the problem using memoization. Assume that we have an array A, where $A[x]$ stores the longest path from s to x. All elements of A are initially set to -1 except $A[s] = 0$.

Algorithm 5 LONGEST(x)

```
for ((v, x) ∈ E) do
  if (A[v] = −1) then
    LONGEST(v)
  if (A[v] + w(v, x) > A[x]) then
    A[x] = A[v] + w(v, x)
```

To solve the longest path problem from s to f, run LONGEST(f). Every node is the argument

of LONGEST only once, and the runtime for processing the node is proportional to the number of edges pointing to it. The total runtime is hence $O(|E|)$.

If there are a small number of subproblems, memoization or dynamic programming can often be used to obtain efficient algorithms. Even if the number of subproblems is large, it is often useful to store solutions that have been computed using as much memory as possible. Note that for these problems, the states usually form the subproblem i.e. small state space gives small number of subproblems. This is not unusual as they usually contain all the information required to complete the construction of the desired object.

2.8 Randomization

When the goal states are very common, the chances of selecting the goal states by randomly selecting a state is high.

Consider the Ethernet, with its *carrier sense multiple access with collision detection* (CSMA/CD) scheme. In the Ethernet, all users share the same communication medium (co-axial cable). In the CSMA/CD scheme, the transmitter listen at the same time that it is transmitting. If some other user is transmitting, it immediately stop transmitting, sends a jam signal and wait a *random* amount of time before starting to transmit again. As long as the number of users is not too high, waiting for a random period is enough to ensure that the next time the user transmit, it is likely to be successful.

In primality testing (testing whether a number is prime), the Miller-Rabin algorithm, randomly chooses a number k and uses it to test whether a number n is composite. It turns out that if n is a composite number, at least $3/4$ of natural numbers less than n are *witnesses*, i.e. they can be used successfully to show that n is composite. So, if we select the numbers randomly m times, the probability that a number that is not prime passes all m tests is no more than $1/4^m$ which decreases to zero very rapidly.

Randomization is used not to find common objects but also to estimate their count (frequently used for estimating integrals). For example, let $\sigma(x, y)$ be an indicator function for a circle of radius 1 centered around the origin (equal to 1 when (x, y) falls inside the circle). Let $p(x)$ and $p(y)$ be the uniform distributions on $[-1, 1]$. As the area of the circle is π, the integral $\int \sigma(x, y)p(x)p(y)dxdy$ evaluates to $\pi/4$ (the area of the square represented by $x \in [-1, 1], y \in [-1, 1]$ is 4). A simple method of estimating the value of π is to randomly select a point uniformly in the square and check the fraction of points that falls in the circle. This fraction will converge to its average value which is $\pi/4$. Methods that exploit randomization in this way is often called *Monte Carlo* methods.

Randomization is often used by algorithm designers to ensure that the average case behaviour is controlled by the algorithm and not by the environment the algorithm is operating in. For example, a simple way to run quicksort is to pick the first element as pivot each time. In this case, the average runtime for a randomly permuted sequence is $O(n \log n)$ but the worst-case run-time for quicksort is $O(n^2)$. Picking the first element as pivot depends on the environment to provide the lists in random order for quicksort to achieve good average case runtime. However, it is possible that the worst case sequences are very common in the environment that quicksort is deployed in, resulting in poor average performance. In *randomized* quicksort, the pivot is selected randomly by the algorithm itself and does not depend on the order in which the data appears. Hence, the algorithm

effectively does a random permutation of the sequence while it is running and its average runtime is unaffected by the environment that it is deployed in.

2.9 Symmetry and Invariance

Consider the 8-queens problem. To check whether a board contains a solution we only need to run CONTAINSQUEENSOL on the four left most columns of the first row. By *symmetry*, a solution can always be found by starting from those position if a solution can be found by starting from any the the last four columns.

Now consider the case where the order of the variables (rows) are not fixed and we store the states that have been fully explored. To represented a state we could have used a list of queen positions on the board. Let's label each board position from 1 to 64. Note that as a list, there are two possible ways to represent queens at position 1 and 12: (1,12) and (12,1). Treating these two lists as different will result in doing more search than necessary - this could be substantially more as n queens can be ordered in $n!$ ways. We say that the states are *invariant* under permutation of the list elements. One way to exploit this symmetry is to use a set (implemented using a sorted list or hash table) instead of a list and check whether the set has been explored before.

In general, you should think about whether the representation you are using is invariant to the transformations that leave the results of the computation unchanged. Ensuring that it is can lead to substantial computational savings. In fact, ensuring invariance to transformations are fundamental problems that have not been satisfactorily solved in many application areas. For example, in face recognition, we would like to recognize a face as the same regardless of translation and rotation in the image, rescaling, and change in lighting. In natural language processing, we would like to know if one sentence is a paraphrase of another sentence (i.e. it means the same thing even though it is written in a different way).

2.10 Exercise

1. Given a file containing at most 4 billion 32-bit integers, you are required to find one that is not in the file. You can use external disk files but only a few hundred bytes of main memory. (Bentley [1])

2. Words that are permutation of each other are called anagrams. For example *pots* and *stop* are anagrams because they are permutations of each other. Find all sets of anagrams in a dictionary. (Bentley [1])

3. Consider the traveling salesman problem. Given a partial tour, argue that the minimum spanning tree through all the remaining nodes is an admissible heuristic function for the cost of completing the tour.

4. Describe an efficient algorithm for finding the length of the longest monotonically increasing subsequence in a sequence of n numbers. For example, in the sequence $S = (9, 5, 2, 8, 7, 3, 1, 6, 4)$, the longest increasing subsequence has length 3 and is either $(2, 3, 4)$ or $(2, 3, 6)$.

5. Consider the following data compression technique. We have a table of m text strings, each of length at most k. We want to encode a data string D of length n using as few text strings as possible. For example, if our table contains $(a, ba, abab, b)$ and the data string is $bababbaababa$, the best way to encode it is $(b, abab, ba, abab, a)$ a total of five code words. Give an $O(nmk)$ algorithm to find the length of the best encoding, You may assume that the string has an encoding in terms of the table.

6. Consider the WALKSAT algorithm for solving SAT problem.

Algorithm 6 WALKSAT(f)

Randomly select an initial assignment $x_1, \ldots, x_n$.
for i = 1 to *maxFlip* **do**
  randomly select an unsatisfied clause c
  Toss a coin probability p
  if *head* **then**
    randomly flip a variable in c
  else
    flip the variable in c that reduces the number of unsatisfied clauses the most
  if f is satisfied **then**
    return true
return false (or rstart with another random initial assignment)

WALKSAT is practically quite successful. Discuss why.

7. (a) In graph colouring, a solution remains a solution if we permute the colours, i.e. if we change *red* to *blue*, *blue* to *green* and *green* to *red*. How would you represent the state for the graph colouring problem to exploit this invariance?

   (b) Consider storing a state and checking whether it has been searched before in problems such as graph colouring. What is the likely limitation when trying to scale the approach to larger problems?

8. Consider an instruction pipeline in a computer that breaks up an instruction into several smaller pieces in order to exploit parallelism. Typically an instruction pipeline may consist of stages that do instruction *fetch*, *decode*, *execution*, and *memory access*. Each stage depends on the previous stage, so the critical path is 4 and the work is also 4 and the parallelism is 1. So, how does a pipeline exploit parallelism?

9. Thomas Edison asked a new researcher to compute the volume of an empty light bulb shell. After a few hours of measurements and calculus, the new researcher returned with an answer of 150. In a few seconds, Edison did his own computation and responded "closer to 155". How did Edison do it? (Bentley [1])

References

[1] J. Bentley. *Programming Pearls*. Addison-Wesley Professional.

[2] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. Introduction to Algorithms.

[3] J Dean. Software engineering advice from building large-scale distributed systems, 2007. Stanford CS295 class lecture `http://research.google.com/people/jeff/stanford-295-talk.pdf`.

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

[5] R. Pike. Notes on Programming in C. *`http://www.lysator.liu.se/c/pikestyle.html`*, 1989.

[6] S.J. Russell and P. Norvig. *Artificial intelligence: a modern approach*. Prentice hall, 2009.