CS2309 CS Research Methodology

Solution

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

Semester 1, 2011/12

Having formulated a problem, we usually need to come up with some sort of a solution. Look at

- Some rules-of-thumb for implementation
  - Useful for building systems
- Algorithms and heuristics for search
  - Useful for finding solution to models
  - Also useful for doing proofs by hand

Outline

- Implementation Tricks
  - Measure before Tuning
  - Use Simple Algorithms
  - Data Dominates
  - Plan to Throw One Away
  - Back of the Envelope Calculations
  - System Efficiency Tricks

Measure before Tuning

Modified from Rob Pike (Pike 1989):

Rule 1: You can't tell where a program is going to spend its time.

- Bottlenecks occur in surprising places.
- Find them first.

Rule 2: Measure.

- Don't tune for speed until you've measured.
- Even then don't unless one part of the code overwhelms the rest.

Use Simple Algorithms

Rule 3: Fancy algorithms are slow when n is small.

- 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.

Rule 4: Fancy algorithms are buggier than simple ones.

- They're much harder to implement.

Data Dominates

Rule 5: Data dominates.

- If you've chosen the right data structures and organized things well, the algorithms will almost always be self evident.

Rule 6: There is no Rule 6.

Plan to Throw One Away

- You will throw one away anyway.
- Lots of innovations in research project.
- Ideas can be reused.

Back of the Envelope Calculations

Some useful numbers and an example (Dean 2007)

- 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
```

System Efficiency Tricks

These are mostly from (Lampson 1983).

- Handle normal and worst case separately:
  - Make sure normal case is fast.
  - Worst case are rare. Okay as long as it works.
  - Don't penalize normal case to improve worst case.
- Use dedicated resources:
  - Dedicated instead of sharing.
  - Example: CPU and graphics card
  - Exploit parallelism.

- Cache answers:
  - Store expensive computation.
  - Save on recomputation.
  - Example:
    - Computer systems have a fast memory (e.g. main memory) which is small and slow memory (e.g. disk) which is large.
    - Fast memory is used as temporary storage to store commonly used items.
  - Which item to replace when fast memory full?
    - temporal locality: favoring recently used items (similar time of access) leads to least recently used (LRU).
    - spatial locality: items physically nearby in memory likely to be accessed together leads to fetching whole block instead of one item.

- Safety First:
  - Cannot predict loads of general purpose systems well
  - Poor performance when load exceed 2/3 capacity
  - Avoid disaster rather than optimize
  - Example:
    - In paging system, only important thing is usually to avoid thrashing.

- Shed Load:
  - Better to shed load to control demand
  - Nothing works when overloaded
  - Processed refused service can do other things in mean time
    - Preferable to waiting
  - Networks discard packets
  - Interactive systems refuse new users.

Outline

- Search for Solution
  - State Representation
  - Divide and Conquer
  - Most Constrained Variable Principle
  - Exploiting Bounds
  - Relaxation
  - Greedy Heuristic
  - Memoization
  - Randomization
  - Symmetry and Invariance

State Representation

Before we search for a solution, we need a representation to search over.

State

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.

States or representations related to the state of computation are useful things to keep in mind.

Examples:

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.

8-Queen

The position of the queens already placed on the board can be a state.

- The order in which the queens were placed is irrelevant.

Shortest Path

The town that you are current at can be the state.

- How you get there is irrelevant to how you can get to the destination

States contain most of the information required for search

- May need to augment a little
- But good place to start.

Divide and Conquer

One of the most common technique:

- Divide the problem into smaller sub-problems.
- Solve the sub-problems.
- Combine the solutions of the sub-problems into a solution of the larger problem.

For this to be effective:

- Divide and combine steps must be efficient.
- It should be possible to solve the sub-problems independently.

Often use recursive solution.

Example:

Four Queens

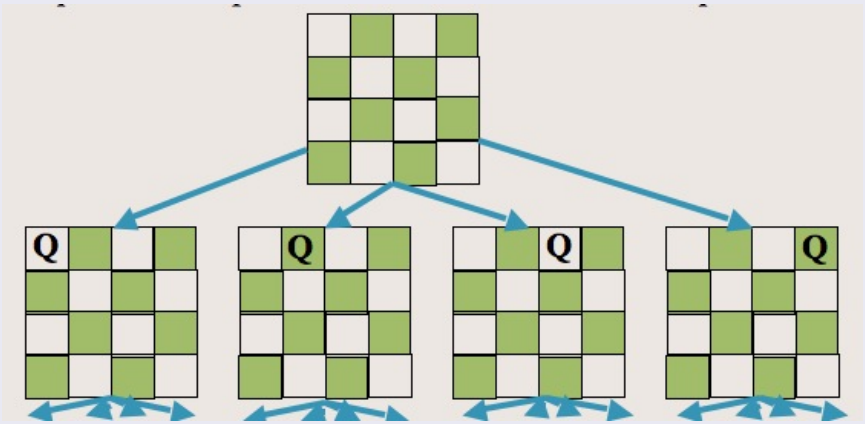
- Place 4 queens on the board so that they do not threaten one another.
- Representation for search?

- Consider state representation principle ...
- Board with currently placed queens, denoted *board*, is the state.

Algorithm 1 ContainsQueenSol(*board*, *row*, *col*)

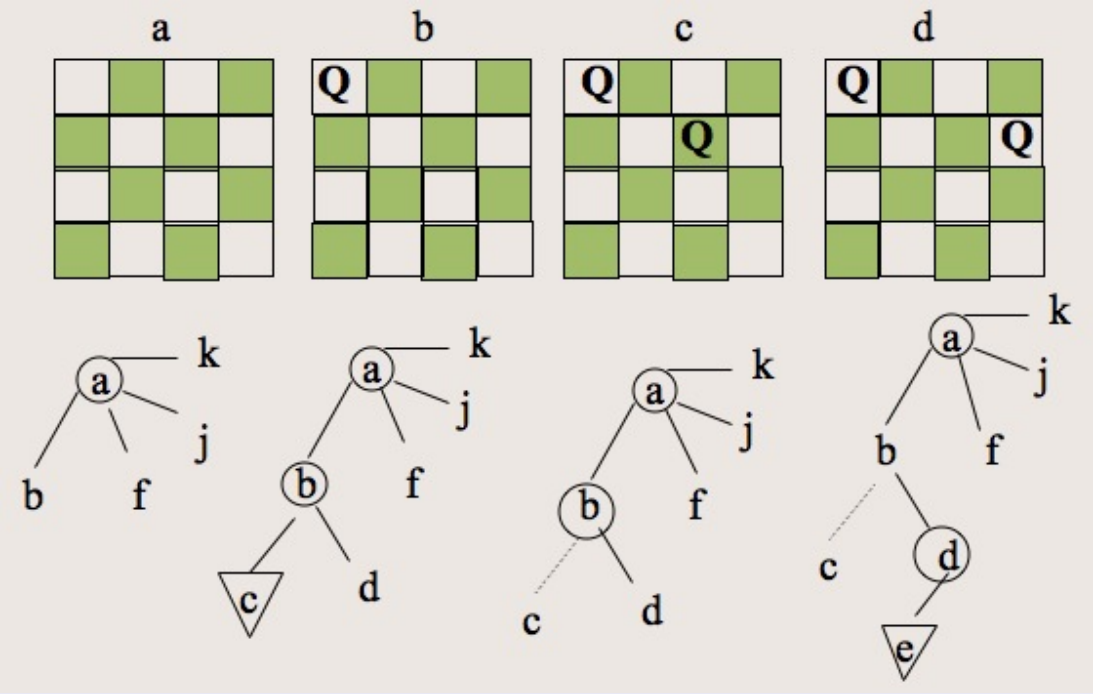
```
Place queen at position (row, col) on board
if row = n then
    return IsLegal(board)
else
    for i = 1 to n do
        if ContainsQueenSol(board, row + 1, i) then
            return true
        end if
    end for
    return false
end if
```

- Use ContainsQueenSol to check whether placing a queen on each column of first row leads to solution.
- Each of these is recursively split further.



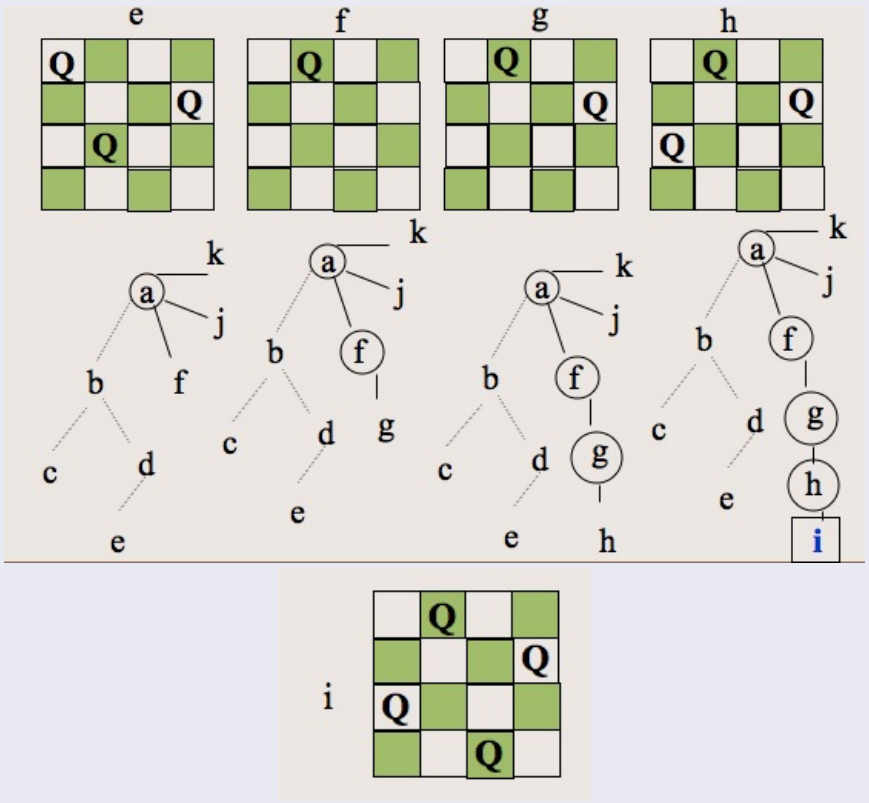
- Problem size is divided roughly equally each time - usually a good idea.

- Can often improve search.
- Prune away node c because cannot place any queen in the following row.
- Backtrack and look at node d.



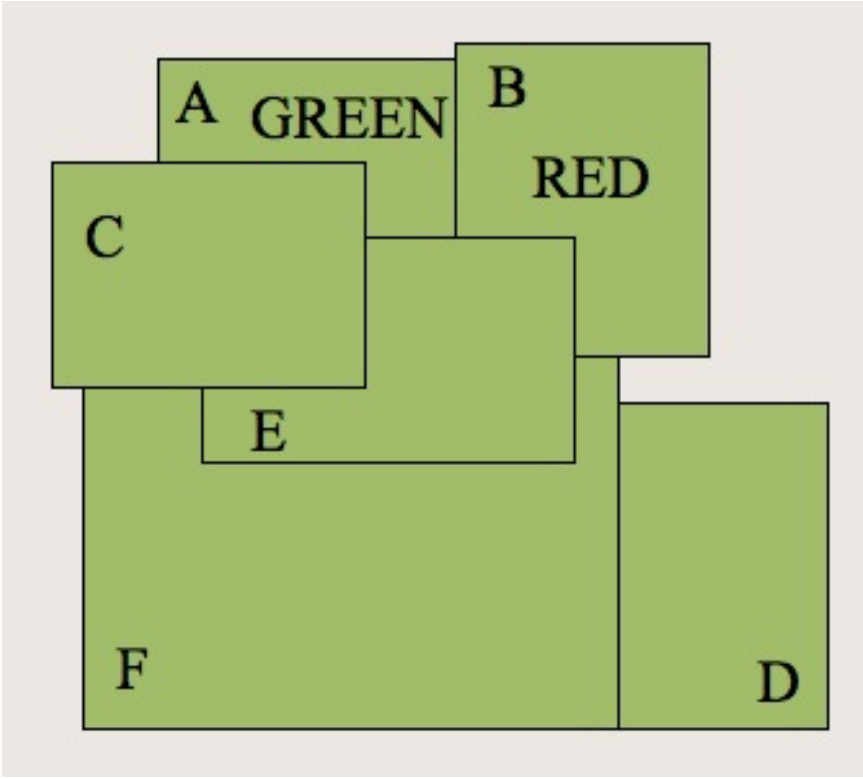
- Often called backtracking search.

- Similarly can prune away node *e* and backtrack.
- Try node *f* next. Go on to *g*, *h* and *i*. Success!
- Prune greatly reduced the amount of search required compared to generating all possible instances of 4-queens and testing them.



Most Constrained Variable Principle

- Consider the map colouring problem shown where countries that have common boundaries cannot be coloured the same.
- Which country should be coloured next assuming we can use at most 3 colours?
- Commonly used heuristic: Select the variable that is most constrained
  - For example: $f(E) = 1$, $f(C) = 2$, $f(F) = 2$, $f(D) = 3$, where f measures the number of remaining options.
  - If fail, fail early. If successful, remaining search space much smaller.



- Colouring *E BLUE* leads to *C RED* and *F GREEN* and *D RED* Solution!!

Example: SAT

- Satisfiability: Is 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.
  - Often represent the boolean formula in CNF form e.g. $(x \vee \neg y) \wedge (y \vee z) \wedge (\neg x)$.
  - How do you implement the *most constrained variable* heuristic for CNF satisfiability?

- Davis-Putnam: successful SAT algorithm
- State is partial assignment of variables
- Branch on values of each variable $x_1, \ldots, x_n$
- Pruning:
  - True if every clause is already true even with a partial assignment of variable
  - False if one clause evaluates to false. Backtrack.
- Most constrained variable heuristic: *unit clause heuristic*.
  - unit clause is a clause with one literal
  - variable is constrained to take the value to make the literal true
  - can cause other clauses to become unit clauses and the forced assignment is continued: *unit propagation*

Exploiting Bounds

- Consider a minimum dominating set problem.
- State is a subset of selected vertices.
- Can prune off the search tree and backtrack
  - if the current state contains more vertices than the best solution so far
  - can do better?

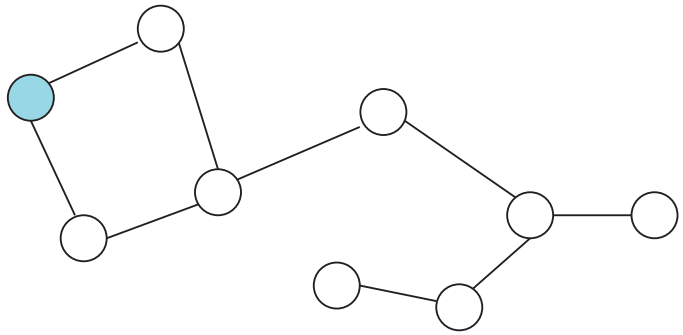


Figure: The shaded vertex dominates the two vertices connected to it. The remaining subgraph requires three additional vertices to dominate.

- Remove the subset already dominated.
  - Number of remaining vertices: N
  - Largest number that can be dominated in subgraph by single vertex: d
  - N/d gives an optimistic estimate (bound) on number of additional vertices required.
- If current subset plus optimistic estimate poorer than best so far, prune.

Branch and Bound

- Technique known as branch and bound
  - Divide current set into subset.
  - For each subset, provide optimistic estimate
  - Prune if optimistic estimate poorer than best solution found so far.
  - Backtrack

A* Search

- Squeeze more out of the bound?
- Can use it to order the nodes so that node that looks best in terms of bound searched first.
- Type of greedy search.
- Called A* search.

Algorithm 2 BestFirstSearch(G, S)

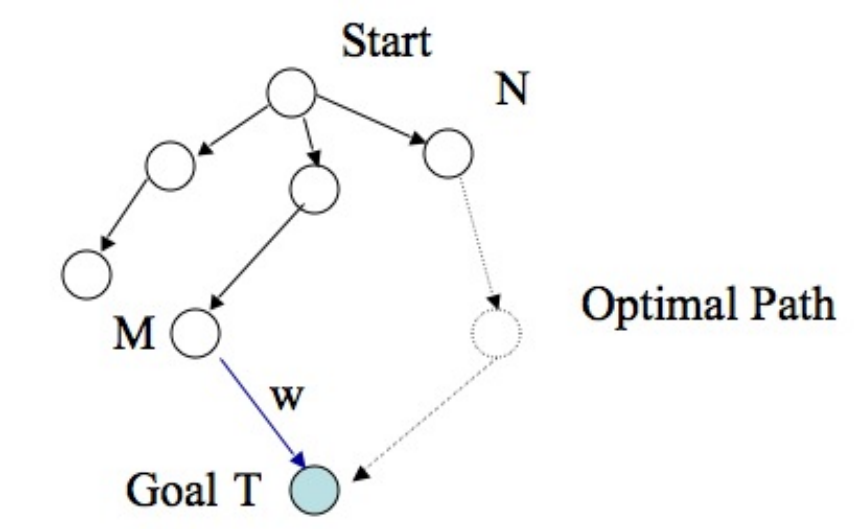
$found = \phi$, $queue = \{S\}$
while $queue \neq \phi$ **do**
  $n =$ ExtractBest$(queue)$
  if IsGoal(n) **then**
    return Cost(n)
  end if
  for $c \in$ Neighbour(n, G) **do**
    UpdateCost(c)
    if $c \notin found$ **then**
      Insert$(c, queue)$
      Insert$(c, found)$
    end if
  end for
end while

A* Search

- Often used to find not just a configuration but path to goal as well: shortest path from S to the goal node T.
- Let $f(n) = g(n) + h(n)$ where
  - $g(n)$ is the cost of the path to the current node.
  - $h(n)$ is an estimate of the cost to the goal node (e.g. the straight line distance in the road map problem)
- If $h(n)$ is optimistic, i.e. it never overestimate the cost to the target node, then BestFirstSearch using $f(n)$ will find the optimal path.

- This algorithm is called A*.
- We call heuristic functions that are optimistic admissible heuristic functions
- An additional condition, consistency, $h(n) \leq h(n') + cost(n, n')$ for every successor n' of n is sufficient to ensure that previously expanded nodes in *found* never need to be expanded again (as shown in the pseudocode).
- For the road map problem, the straight line heuristic is admissible and consistent.

A* Illustration



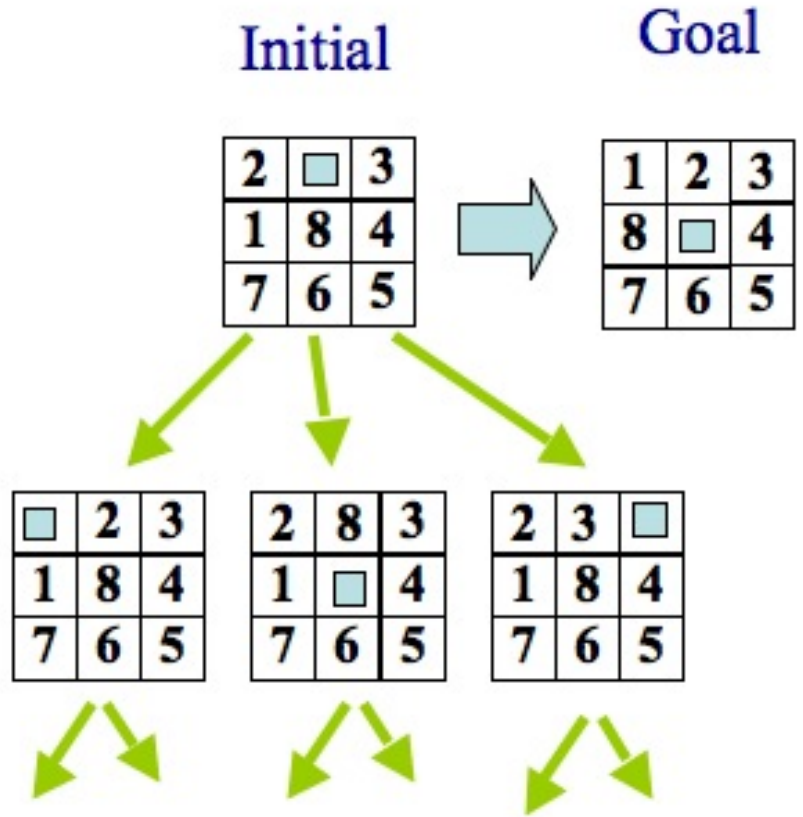
- N lies on the optimal path, so $f(N) = g(N) + h(N) < g(M) + w$, since h is optimistic.
- N will be chosen for expansion ahead of T. This will be true as long for any node (like M) that is connected to T but not on the optimal path to T.
- Once T is selected, cannot have better path to goal.

Relaxation

Eight Puzzle

Two admissible and consistent heuristics:

- h_1: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
- h_2: Sum of distances of tiles from their goal position. Admissible because each move can only move one tile one step closer to its goal



- Heuristics can be discovered by consulting simplified models of the problem domain. Relax the constraints to obtain simplified models
- 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.
- If we can move one square in any direction even onto an occupied square then h_2 gives the number of steps required.

Greedy Heuristic

- Being greedy i.e. take the step that currently looks the best according to some measure of goodness, is a natural problem solving heuristic. Examples:
  - The A* algorithm is a greedy algorithm.
  - Interval graph coloring example shown in earlier lecture.
  - If we want to replace an item in a cache, we often replace the least recently used item.
  - If we want to maximize the function $f(x)$ with respect to x, 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.
- Greed is often good, but not always optimal: special properties are required for optimality.

Suboptimality of Greedy Coin Changing

Need change for n cents. Use the fewest coin possible. There are k denomination $d_1, \ldots, d_k$.

- Always picking the largest possible coin (greedy) does not always work:
  - Available denomination: 25, 10, 1.
  - Make change for 30:
  - Greedy would pick 25+1+1+1+1+1.
  - Optimal is 10+10+10.

Memoization

- In some problems, the same sub-problems appears many times.
- Memoization is based on the idea of storing the solution of sub-problems and looking them up rather than recalculating them.

Coin Changing

- $M[i]$: min coins needed to make i cents.
- Denominations $d_1, \ldots, d_k$.
- $M[i]$ satisfies the following relationship:

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

Define an array $M[0, n]$. Initialize $M[0] = 0$, other elements to ∞.

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
        end if
    end for
    M[i] = minCoin
    return minCoin
end if
```

- Consider what happens in a normal recursive program i.e. without
  else if $M[i] \neq \infty$ **then**
  return $M[i]$
- NumCoins calls itself k times.
- Each child routine in turn calls itself k times.
- until $i < 0$
- Runtime exponential in n for n cents.

- Now consider what happens in the memoized version i.e. with
  else if $M[i] \neq \infty$ **then**
  return $M[i]$
- Execution graph of the problem is acyclic
  - To solve for $M[i]$, only need to solve for strictly smaller than i.
  - Hence each $M[i]$ solved only once.
- Each $M[i]$ calls NumCoins at most k times
  - The part of the code that test whether $i < 0$ or whether $M[i]$ has been solved before is run at most nk times.
- Remaining part of the code is done once for each subproblem - call NumCoins k times, also $O(nk)$ time.
- So total time $O(nk)$.

Program can also be written in an iterative manner: dynamic programming.

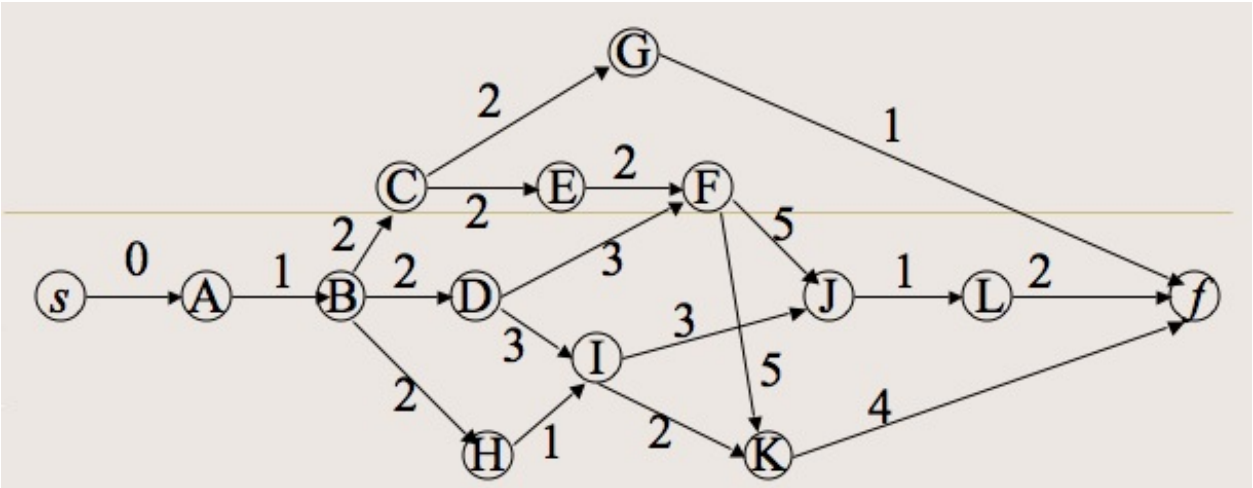
Algorithm 4 NumCoins(n)

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 $(i - d[j] \geq 0)$ and $(M[i - d[j]] + 1 < M[i])$ **then**
      $M[i] = M[i - d[j]] + 1$
    end if
  end for
end for

Example: $n = 9$, 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

One optimal solution: $9 = 3 + 3 + 3$.



Longest path to G, P(G) = 5 Longest path to E, P(E) = 5
Longest path to D, P(D) = 3 Longest path to H, P(H) = 3
Longest path to F, P(F) = max(P(E)+2,P(D)+3)=7
Longest path to I, P(I) = max(P(D)+3,P(H)+1)=6
Longest path to J, P(J) = max(P(F)+5,P(I)+3)=12
Longest path to K, P(K) = max(P(F)+5,P(I)+2)=12
Longest path to L, P(L) = P(J)+1 = 13
Longest path to f, P(f) = max(P(G)+1,P(L)+2,P(K)+4)=16

Solve longest path problem using memoization.

- $A[x]$ stores the longest path from s to x.
- A are initially set to -1 except $A[s] = 0$.

Algorithm 5 LONGEST(x)

for $((v, x) \in E)$ **do**
  if $(A[v] = -1)$ **then**
    LONGEST(v)
  end if
  if $(A[v] + w(v, x) > A[x])$ **then**
    $A[x] = A[v] + w(v, x)$
  end if
end for

Total runtime $O(|E|)$

Randomization

When the goal states very common, randomly select and testing a state works .

Ethernet

- Carrier sense multiple access with collision detection (CSMA/CD) scheme
- All users share the same communication medium (co-axial cable)
- Transmitter listen while transmitting.
- If some other user is transmitting
  - immediately stop transmitting
  - send a jam signal
  - wait a *random* amount of time
  - start transmitting again.
- As long as the number of users is not too high, waiting for a random period likely to be successful.

Primality testing

- Miller-Rabin algorithm for testing whether n is prime.
- If n is composite (not prime), at least $3/4$ of natural numbers less than n are witnesses - can be used in a simple test to show n composite.
- Randomly choose number $k < n$ and test.
- Do m times.
- Probability that a composite number passes all m test is no more than $1/4^m$.

Another effective use of randomization is to count instead of to find. Frequently used to estimate integrals.

Estimating π

- $\sigma(x, y)$: indicator function for circle of radius 1 centered around the origin
  - equal to 1 when (x, y) falls inside the circle.
  - zero otherwise
- $p(x)$ and $p(y)$: uniform distributions on $[-1, 1]$.
- Area of circle with radius 1 is π, so
  - $\int \sigma(x, y)p(x)p(y)dxdy = \pi/4$
- Randomly select points using $p(x)p(y)$ and check fraction of points that fall in circle.
- Fraction will converge to $\pi/4$
- Known as Monte Carlo method.

Symmetry and Invariance

- Considering symmetries and invariances can lead to considerable computational savings.
- In 8-queens problem
  - running ContainsQueenSol on the four left most columns of the first row is sufficient.
  - To test whether two states are the same
    - Set of queen positions to represent state - okay
    - Sorted list of queen positions to represent state - okay
    - Unsorted list of queen positions to represent state - poor as it fails to capture invariance under permutation of elements.

References

1. Notes on Programming in C. *R. Pike* `http://www.lysator.liu.se/c/pikestyle.html`, 1989.
2. Software engineering advice from building large-scale distributed systems, 2007. *J. Dean.* Stanford CS295 class lecture `http://research.google.com/people/jeff/stanford-295-talk.pdf`.
3. Hints for computer system design. *B.W. Lampson.* ACM SIGOPS Operating Systems Review, 17(5):3348, 1983.