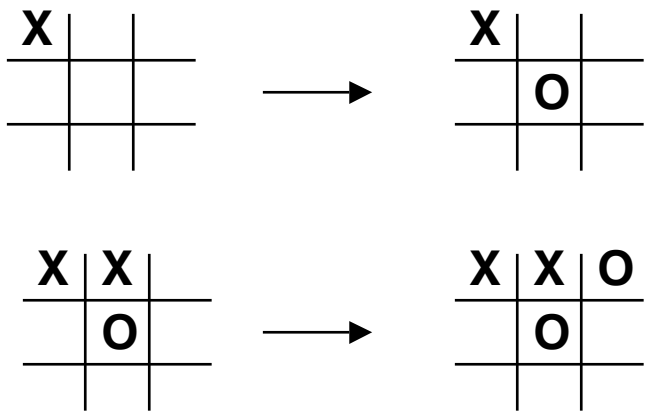
State Space

- Many possible ways of representing a problem
- State Space is a natural representation scheme
- A State Space consists of a set of ‘states’ which can be thought of as a snapshot of a problem
  - all relevant variables are represented in the state
  - each variable holds a legal value
- examples from the Missionary and Cannibals problem (what is missing?)

Counter Example: Not Using State Space to Solve the Problem

- Solving Tic Tac Toe using a data base look up for best moves
- e.g. Computer is 'O'



Each Transition Pair is recorded in a data base

Input **Best Move**

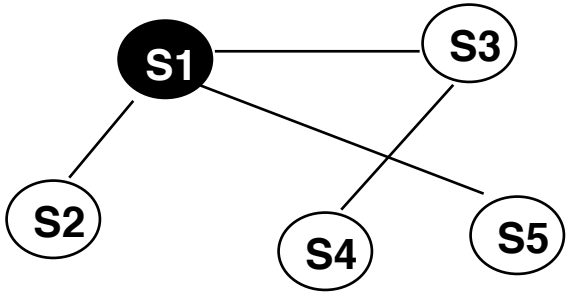
- Simple but unfortunately most problems have exponential numbers of such rules

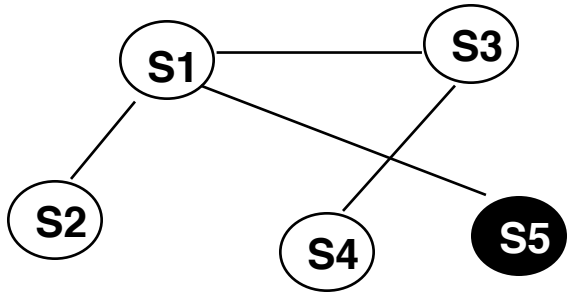
Production Systems

- A set of rules of the form pattern $\longrightarrow$ action
  - The pattern matches a state, and the action changes the state to another state
- A task specific database of current knowledge about the system (current state)
- A control strategy that specifies the order in which the rules will be compared to the database with conflict resolution

State Space as a Graph

- Each node in the graph is a possible state
- Each edge is a legal transition that can transform the current state into the next state





- **Therefore the solution of the problem becomes a search through the state space**

Goal Of Search

- Sometimes solution is some final state
- Other times the solution is a path to that end state

Solution as end state:

- Traveling Salesman Problem
- Chess
- Graph Colouring
- Tic Tac Toe
- N Queens

Solution as path:

- Missionaries and Cannibals
- 8 puzzle
- Towers of Hanoi

Depth First Search

DFS (S):

1. If S is a goal state, quit and return *success*
2. Otherwise, do until *success* or *failure* is signaled:
   - Generate state E, a successor of S. If no more successors signal *failure*
   - Call DFS (E)

- Almost the same as a depth first tree traversal except
  - all nodes generated on the fly by production system
  - algorithm halts when solution found
- DFS assumes tree structure of search space; may not be true
  - If not, can get caught in cycles -> DFS must then be modified
    e.g. each state has a flag that is raised when node is visited

Breadth First Search

BFS (S):

1. Create a variable called NODE-LIST and set it to S
2. Until a goal state is found or NODE-LIST is empty do:
   - Remove the first element from NODE-LIST and call it E;
     If NODE-LIST was empty quit
   - For each way that each rule can match the state E do:
     - » Apply the rule to generate a new state
     - » If new state is a goal state quit and return this state
     - » O.W. add the new state to the end of NODE-LIST

Changing a Cyclic Graph Into a Tree

- Most production systems include cycles
- Cycles must be broken to turn graph into a tree to use the above tree searching techniques
- Can't "mark" the nodes since they are generated dynamically
- Therefore a list is kept of all nodes that have been visited (called the "Closed" list)
- Each node examined is first checked against the "Closed" list, if it is in it the node is ignored and the next node is examined

Algorithm to Break Cycles

- When a node is examined
  - check node to see if it is in “Closed” list
  - if node is in the list
    - » ignore it
  - else
    - » add node to “Closed” list
    - » process node

Example: DFS with cycle cutting

Initializations: S = first state, CLOSED = empty list

DFS (S):

If S is in CLOSED

return *failure*

Else

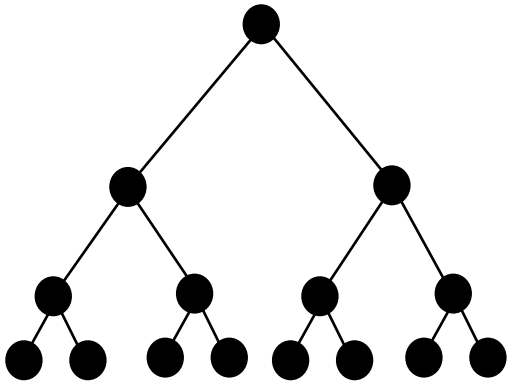
Place S in CLOSED

If S is a goal state, return *success*

Loop

- Generate state E, a successor of S.
  - If no more successors return *failure*
- result = DFS (E)
- if result = *success* return *success*

BFS vs. DFS



- BFS expensive wrt space
  - linear in # of nodes
- DFS only stores a max of log of the # of nodes

- BFS constant memory needed
- DFS linear in # of nodes

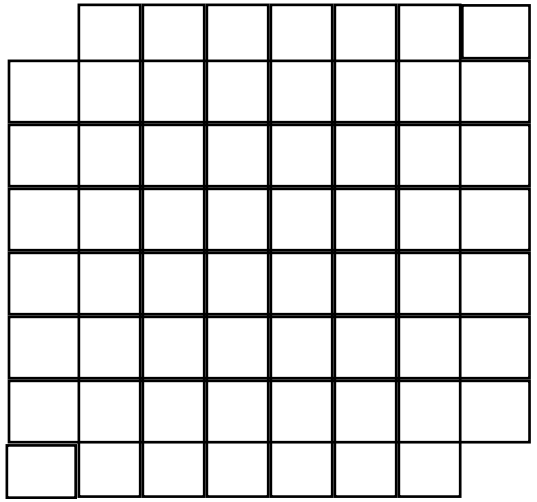
- Time to find sol^n depends on where the sol^n is in the tree
- DFS may find a longer path than BFS when multiple sol^{n}s exist
- BFS guaranteed minimum path solution

Knowledge in Representation

- Representation of state-space can affect the amount of search needed
- problem with comparisons between search techniques if representation not the same

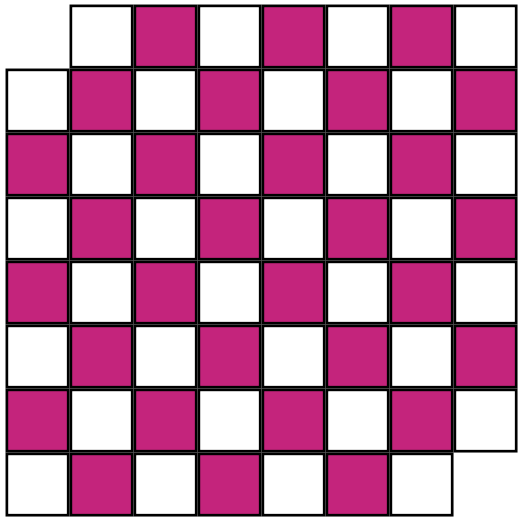
Representation Example

- Mutilated chess board
  - corners removed from top left and bottom right
- Can you tile this board with dominoes that cover two squares?



Representation 1

Representation Example Continued



Representation 2

Number of White Square = 32

Number of Black Square = 30

Representation 3

Heuristic Search

- Problem with DFS and BFS -> there is no way to guide the search
- Solution can be anywhere in tree. In the worst case all possible states will be traversed

- One ‘solution’ to this problem is to look in the search space where the final state is likely to be
- This of course will be problem specific
- A function is usually created that evaluates how good the current solution is and this function is used to help guide the search process

- This guided search called a heuristic search

A Heuristic

- Derived from the Greek word *heuriskein*: “to find”; “to discover”
- Has been used (and is sometimes still used) to mean “A process that may solve a given problem, but offers no guarantees of doing so” Newall, Shaw, & Simon 1963
- Heuristics can also be thought of as a rule of thumb
- Can refer to any technique that improves average-case performance but not necessarily worst-case performance
- We use it here as a function that provides an estimate of solution cost

Hill Climbing

Simple-Hill-Climber (S)

- Evaluate S; If goal state return and quit
- Loop until a solution is found or no neighbors left
  - look at next neighbor NN
  - Evaluate NN
    - » If NN is goal return and quit
    - » If NN is better than S, S := NN (and reset neighbors)

Steepest-Ascent-HC (S)

- Evaluate S; If goal state return and quit
- SUCC := S
- Loop until a solution is found or no neighbors left
  - For all neighbors (NN) of S
    - » evaluate NN
    - » if NN is goal then return NN and quit
    - » if NN is better than SUCC then SUCC := NN
  - If SUCC is better than S then S := SUCC (and reset neighbors)

Hill Climbing Continued

Stochastic-Hill-Climber (S)

- Evaluate S; If goal state return and quit
- Loop until a solution is found or no neighbors left
  - look at some random neighbor RN
  - Evaluate RN
    - » If RN is goal return and quit
    - » If RN is better than S, S := RN (and reset neighbors)

Problems with Hill Climbing

- Hill Climbing will get stuck at local maximums in the search space
- Can get stuck on a ‘plateau’

Solutions

- backtrack to earlier node and force it to go in a new direction
- take a big jump to somewhere else in search space
- Simulated Annealing (next)
- Genetic Algorithms

Simulated Annealing

- Simulate the annealing process of creating metal alloys
- Start off hot, and cool down slowly which allows the various metals to crystallize into a global uniform structure
- If cooled too fast the metals crystallize in pockets
- If cooled too slowly, get a uniform crystallization but wastes time
- Use this idea to try to find global minimum (we are now minimum finding instead of maximum finding — but this is really the same thing) by allowing wandering from the hill-climbing algorithm while system still hot but reducing to hill climbing as system cools

Simulated Annealing Details

- The Probability to move to a higher energy state in physics is where k is the Boltzman constant

$$p = \frac{1}{e^{\Delta E / kT}}$$

- Similarly (when trying to find the minimum) the probability to move to a state with a higher (worse) heuristic function evaluation in SA is

$$p = \frac{1}{e^{\Delta E / T(t)}}$$

where

ΔE = (value of current state) – (value of new state)

T(t) is the temperature schedule (a function of time t)

  - temperature monotonically decreases with time,
  - eventually T reaches 0 at which point the system becomes simple "hill descending"

SA Details When Maximizing

- the probability to move to a state with a lower (worse) heuristic function evaluation in SA is

$$p = e^{\Delta E/T}$$

where

ΔE = (value of new state) – (value of current state)

(The negation of the ΔE used when minimizing)

T(t) is the temperature schedule (a function of time t)

- temperature monotonically decreases with time
- eventually T reaches 0 at which point the system becomes simple "hill climbing"

Simulated Annealing Algorithm

Simulated-Annealing (problem, schedule) From Russell and Norvig

```
current :=  Initial-State(Problem)
for t := 1 to ∝ do
    T := schedule(t)
    if T = 0 then return current
    next := a randomly selected successor of current
    ΔE := Value(next) – Value(current)
    if ΔE > 0 then
        "Always go to a better solution"
        current := next
    else
        "Leave a better solution for a worse one with prob. e ΔE/T "
        current := next only with probability e ΔE/T
```

SA: Meta Heuristics

- If the solution is better always move to it
- If the solution is worse but the slope up (out of the valley) is shallow, try it out
- If the solution is worse but the slope is steep, don't try it out as readily (with an exponentially decreasing probability)
- As time goes on, don't try out worse solutions as frequently (again with an exponentially decreasing probability)

SA Effects

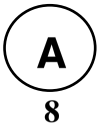
- The probability of ascending to poorer states, or moving along a plateau is large at the beginning of the annealing process (when T(t) is large), so the space can be well searched
  - local minimums can be passed over
  - Ignore steep ascents
    - » This implies that you are in a deep valley, which is assumed to be good
- As time increases the search gets trapped in one Valley and gets stuck there as T(t) becomes small and the probability of getting out of the Valley is too small. SA then becomes 'hill descending' and descends to the bottom of that valley, which is hopefully the global minimum

Best First Search

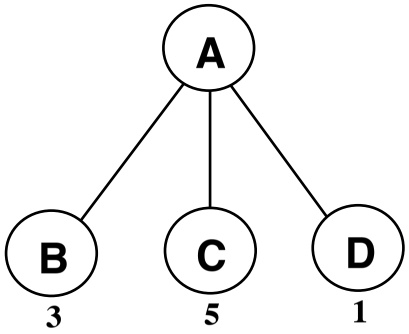
- Combine BFS and DFS using a heuristic function
- Expand the branch that has the best evaluation under the heuristic function
- Similar to hill climbing but can go back to ‘discarded’ branches

Example of BestFS

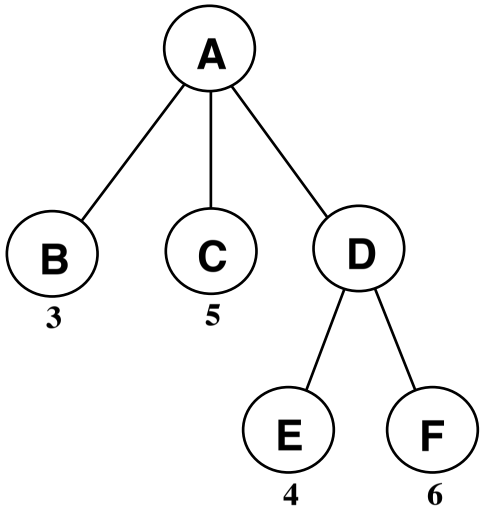
Step 1



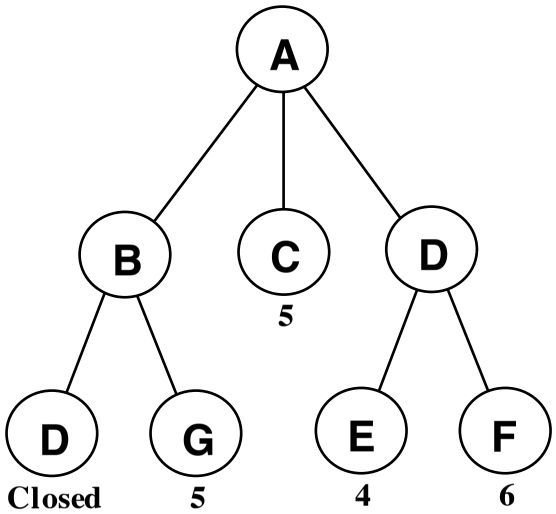
Step 2



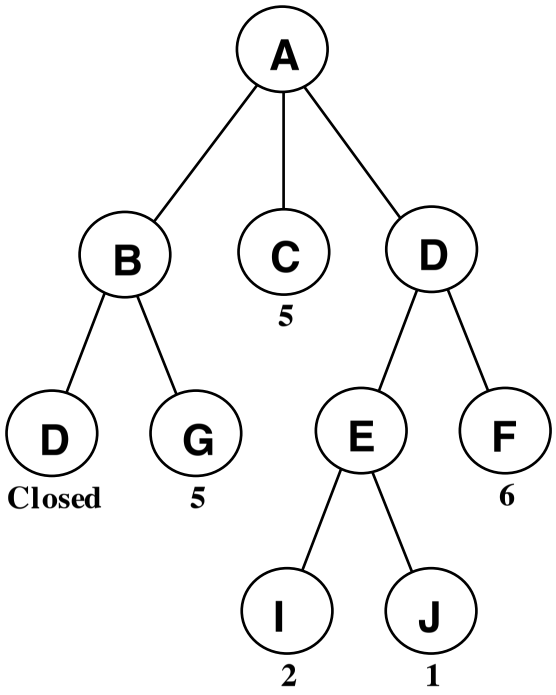
Step 3



Step 4



Step 5



Best First Search Algorithm

- Initialize Open to initial state, Closed to the empty list
- Until a goal is found or no nodes left in Open do:
  - Pick the best node in OPEN
  - Generate its successors, place node in CLOSED
  - For each successor do:
    - » If not previously generated (not found in OPEN or CLOSED), evaluate, add to OPEN

OPEN: generated nodes who's children have not been evaluated yet

- implemented as a priority queue (heap structure)

CLOSED: nodes that have been examined

- used to see if a node has been visited if searching a graph instead of a tree
- same as in DFS and BFS

A* Search

- A modification of the BestFS
- Used when searching for the optimal path
- The heuristic function f(S) is broken into two parts:
  - g(S): The cost of the path so far
  - h(S): The heuristic estimate of cost to end goal
  - f(S) = g(S) + h(S)

A* Algorithm

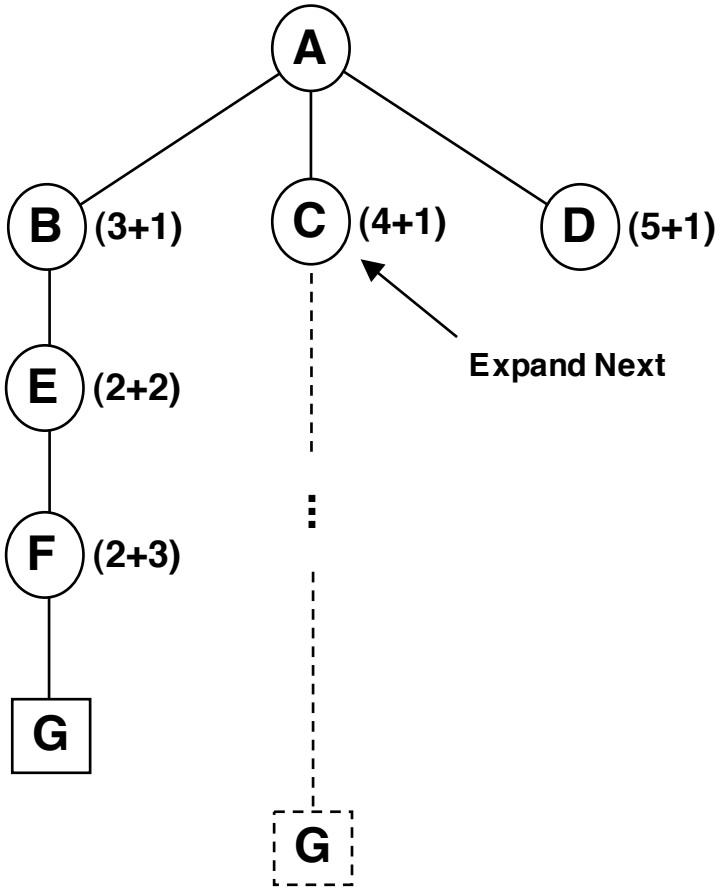
- Initialize OPEN to initial state
- Until a goal is found or no nodes left in OPEN do:
  - Pick the best node in OPEN
  - Generate its successors (recording the successors in a list); place in CLOSED
  - For each successor do:
    - » If not previously generated (not found in OPEN or CLOSED), evaluate, add to OPEN, and record its parent
    - » If previously generated (found in OPEN or CLOSED), and if new path is better then the previous one
      - change parent pointer that was recorded in the found node
    - » If parent changed
      - Update the cost of getting to this node
      - update the cost of getting to the children
        - Do this by recurssively “regenerating” the successors using the list of successors that had been recorded in the found node
      - Make sure the priority queue is reordered accordingly

Properties of A*

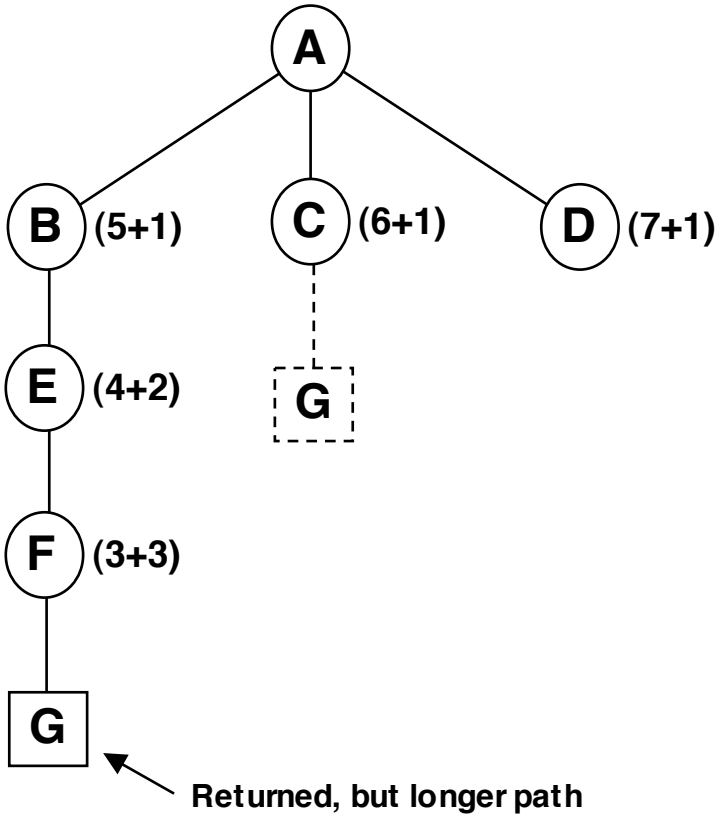
- Becomes simple BestFS if g(S) = 0 for any S
- When a child state is formed g(S) can be incremented by 1, or be weighted based on which production system operator generated the state
- Becomes BreadthFS if g += 1 per generation and h=0 for all states
- If h is the perfect estimator of the distance to the goal (call it H) then A* will immediately find and traverse the optimal path to the solution (no backtracking)
- If h never overestimates H then optimal path to the solution will be found (if it exists)
  - Problem is finding such an h

Under / Over Estimation

h Underestimates H



h Overestimates H



Goal is G

The Importance of the Heuristic Function

- If have exact Heuristic Function H, then the search gets solved optimally
- exact H is usually **very** hard to find (in many cases it would be a solution to an NP problem in polytime, which is probably not possible to compute in less time than it would take to do the exponential sized search)
- next best is to guarantee that h underestimates the distance to the solution, so a minimum path to the goal is likewise guaranteed

Heuristic function vs. search time

- The better your heuristic, the less amount of searching your system will do (improves your average time complexity)
- However, to compute such a heuristic (if you could figure out a good algorithm), usually costs computation cycles that could be used to process more nodes in the search
  - hence the trade-off of complex heuristics vs. more search done

Beam Search

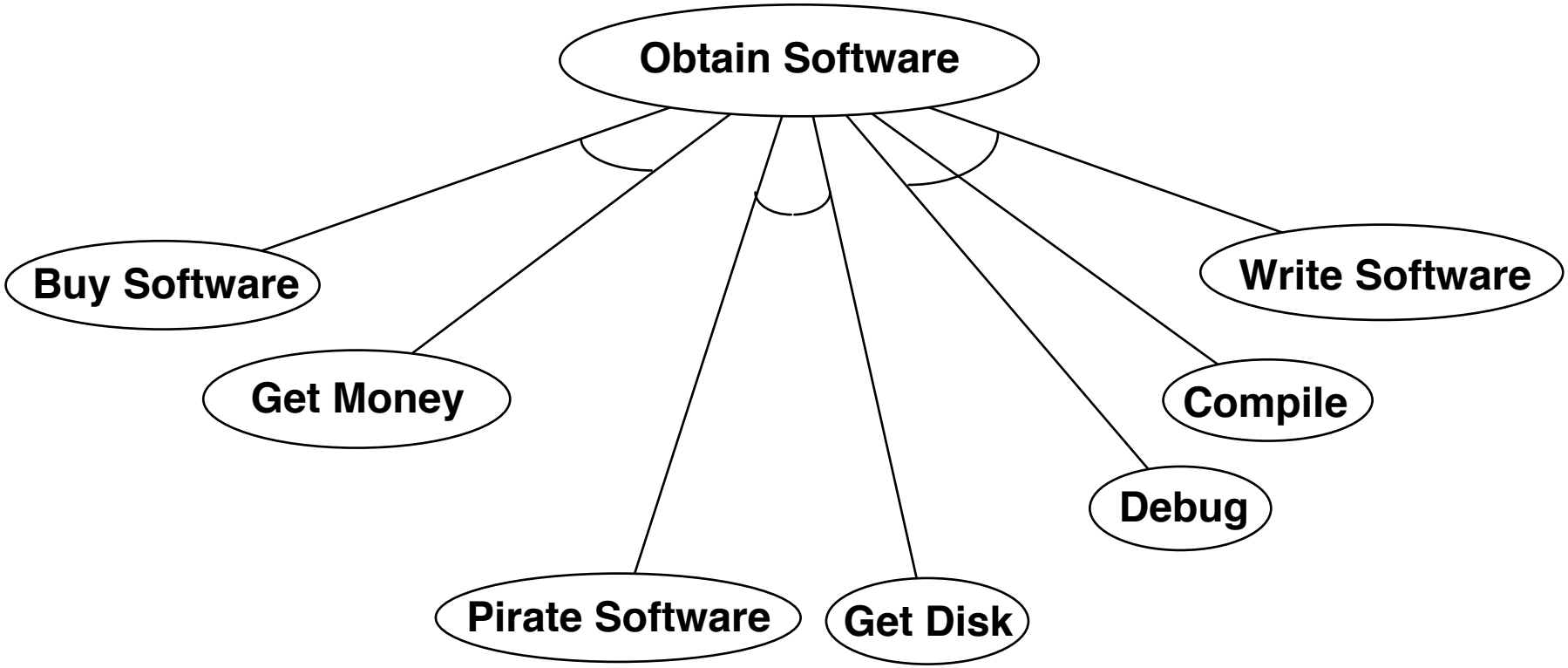
- Same as BestFS and A* with one difference
- Instead of keeping the list OPEN unbounded in size, beam search fixes the size of OPEN
- Now OPEN only contains the best K evaluated nodes
- If new node to add is not better then any in OPEN, and OPEN is full, then the new node is not added
- If the new node is to be inserted in the middle of the priority queue, and OPEN is full, then the node at the end of OPEN (the one with least priority) is dropped

OR Graphs vs. AND-OR Graphs

- In the previous search techniques, the solution can be found down any path independent of any other path
- This is called an OR graph
- However, there may be sub goals that must all be solved for a solution to be found
- Each subgoal is its own subtree, and all subtrees must have its own end state found if the path is to be considered satisfied

Example of an AND-OR Graph

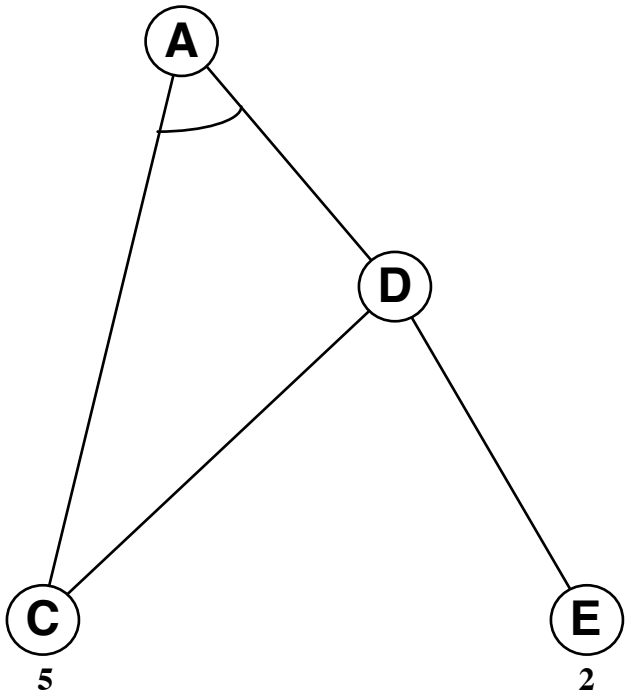
- Getting software to accomplish a task



Problem Reduction Algorithm

- Initialize the graph to the starting node
- Until the starting node is labeled SOLVED or its cost > FUTILITY do:
  - Start at initial node and traverse best path
    - » accumulate a set of notes on the path not expanded or labeled SOLVED
  - pick an unexpanded node and expand
    - » if no successors node cost = FUTILITY
    - » add successors to graph after computing the heuristic function f for each
    - » if f = 0 for any node mark node as SOLVED
  - propagate change back through path
    - » if child is an OR child and is SOLVED mark parent as SOLVED
    - » if AND children are all solved, mark parent as SOLVED
    - » change f estimate as determined by children
    - » as back up the tree change current best path associated with each node (which will be on the original best path) if updated f values warrant it

Interacting Subgoals



Branch and Bound

- if know that current path (branch) is already worse than some other known path, stop expanding it (bound).
- have already encountered Branch and Bound: A* stops expanded a branch if its heuristic value h becomes larger than some other branch

Constraint Satisfaction Problems and Branch and Bound

- Problems where there are natural constraints on the system (fixed resources, impossibility conditions, etc.)
- Constraints are handled by Branch and Bound technique
  - you branch out in your normal search pattern, but stop expanding a branch if it fails a constraint (backtracking may occur when that happens)
- trivial example: in Missionaries and Cannibals, you do not continue to search along a branch if the Cannibals have just eaten some (or all) of the Missionaries

Mini-Max Search

- Search can be used to find the correct move in a two player game
- Realistic alternative to the optimal but exponential algorithm of generating all possible paths and only playing those that lead to a winning final position
- uses finite depth look-ahead with a heuristic function for evaluating how good a given game state is

Mini-Max Continued

- Extend Tree down to a given search depth
- Top of tree is the Computer's move
  - wants move to ultimately be one step closer to a winning position
  - i.e. wants move that maximizes own chance of winning
- Next move is opposition's
  - opposition assumed to perform a move that is best for him/her self
  - i.e. assumed to take move that minimizes computer's chance of winning

Mini-Max Algorithm

```
function MINIMAX-DECISION(game) returns an operator
   for each op in OPERATORS[game] do
      VALUE[op] := MIN-VALUE(APPLY(op, game), game)
   end
   return the op with the highest VALUE[op]

function MAX-VALUE(state, game) returns a utility value
   if CUTOFF-TEST(state,) then return EVAL(state)
   value := - ∞
   for each s in SUCCESSORS(state) do
      value := MAX(value, MIN-VALUE(s, game))
   end
   return value

function MIN-VALUE(state, game) returns a utility value
   if CUTOFF-TEST(state,) then return EVAL(state)
   value := ∞
   for each s in SUCCESSORS(state) do
      value := MIN(value, MAX-VALUE(s, game))
   end
   return value
```

Branch and Bound
Alpha-Beta Pruning

- Branch and Bound: if know that current path (branch) is already worse then some other know path, stop expanding it (bound).
- Alpha-Beta is a branch and bound technique for mini-max search
- If you know that the level above won't choose your branch because you have already found a value along one of your sub-branches that is too good, stop looking at other sub-branches that haven't been looked at yet

Alpha-Beta Pruning Algorithm

- From Russell and Norvig

α = best score for MAX so far
β = best score for MIN so far
game = game description
state = current state in game

```
function MAX-VALUE(state, game, α, β) returns a utility value

   if CUTOFF-TEST(state,) then return EVAL(state)
   for each s in SUCCESSORS(state) do
      α := MAX(α, MIN-VALUE(s, game, α, β))
      if α ≥ β then return β
   end
   return α

function MIN-VALUE(state, game, α, β) returns a utility value

   if CUTOFF-TEST(state,) then return EVAL(state)
   for each s in SUCCESSORS(state) do
      β := MIN(β, MAX-VALUE(s, game, α, β))
      if β ≤ α then return α
   end
   return β
```

Improving Game Playing

- Increase Depth of Search
- Have better heuristic for game state evaluation

Changing Levels of Difficulty

- Increase Depth of Search

Problems with Mini-Max

- Horizon effect – can't see beyond depth
  - do to exponential increase in tree size, only very limited depth feasible
  - attempt to fix: waiting for quiescence
- May want to use look up tables for end games, and opening moves (called Book Moves)