CS 3243 – Algorithms review

Chapters 2-4 and 6

Note: you will want to print this with notes.

Problem-solving agents

```
function SIMPLE-PROBLEM-SOLVING-AGENT( percept) returns an action
   static: seq, an action sequence, initially empty
           state, some description of the current world state
           goal, a goal, initially null
           problem, a problem formulation
   state ← UPDATE-STATE(state, percept)
   if seq is empty then do
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH( problem)
   action ← FIRST(seq)
   seq ← REST(seq)
   return action
```

Note here that the sequence of actions precomputed by the agent.

Subsequent actions are returned off of the queue, without recomputation.

Only when the queue of actions is exhausted does the S-P-S-A compute new moves.

To think about: what does this mean in terms of unexpected results of actions and noisy environments?

Tree search

function TREE-SEARCH(*problem, strategy*) **returns** a solution, or failure
  initialize the search tree using the initial state of *problem*
  loop do
    if there are no candidates for expansion **then return** failure
    choose a leaf node for expansion according to *strategy*
    if the node contains a goal state **then return** the corresponding solution
    else expand the node and add the resulting nodes to the search tree

Tree search is the basic algorithm in which we can change into other uninformed and informed searches based on what methods we use to decide which candidate to expand. Initially only the initial state is in the search tree as a simple. The above figure used in the slides is the English version of the pseudo code version on pg 72.

In both, the current node is checked to see whether it is a goal state. If so, the solution is returned (the path through the tree). Otherwise the node is expanded and its descendents placed into the fringe in some order. The specific order is important and subject to the strategy employed by the agent (e.g., DFS, BFS, etc.).

Note that the ordering the nodes in the fringe can be done in the REMOVE-FIRST, INSERT-ALL stages. The EXPAND function itself cannot do this entirely by itself as it does not have access to the fringe, merely the set of successors that it generates.

Depth-limited search

```
function DEPTH-LIMITED-SEARCH( problem, limit) returns soln/fail/cutoff
    RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)
function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
    cutoff-occurred? ← false
    if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
    else if DEPTH[node] = limit then return cutoff
    else for each successor in EXPAND(node, problem) do
        result ← RECURSIVE-DLS(successor, problem, limit)
        if result = cutoff then cutoff-occurred? ← true
        else if result ≠ failure then return result
    if cutoff-occurred? then return cutoff else return failure
```

Depth limited search calls its recursive partner RECURSIVE-DLS to search the search tree using a depth limit.

RECURSIVE-DLS implements the TREE-SEARCH algorithm using depth first search with a depth limit. You can see the effect of the depth limit in the **else if** statement, the 4th line in the RECURSIVE-DLS method() . Here, if the depth limit is reached, a flag *cutoff* is returned to the caller (either DEPTH-LIMITED-SEARCH or RECURSIVE-DLS). In the latter case, RECURSIVE-DLS backtracks and has to try another successor if possible. In the former (when *cutoff* is returned to the calling function DEPTH-LIMIT-SEARCH, the search tree (to depth *limit*) has been entirely explored and exhausted for goal states.

The depth limit *limit* is propagated in each recursive call and does not change (is invariant). The Depth[node] call in line 4 retrieves the node’s depth and checks it with the limit.

Iterative deepening search

function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution, or failure
  inputs: *problem*, a problem

  for *depth* ← 0 **to** ∞ **do**
    result ← DEPTH-LIMITED-SEARCH(*problem*, *depth*)
    if *result* ≠ cutoff **then return** *result*

ITERATIVE-DEEPENING-SEARCH is a shell around the DEPTH-LIMITED-SEARCH that calls the DEPTH-LIMITED-SEARCH with increasingly large depth limits. It increments the depth limit by 1 each time.

To think about: ITERATIVE-DEEPENING-SEARCH only increases by one each time, which leads to overhead. However, we've shown in class that the overhead can be relatively small compared to the final cost. Still, would increasing this rate be useful in some configurations?

Graph search

```
function GRAPH-SEARCH( problem, fringe) returns a solution, or failure
   closed ← an empty set
   fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
   loop do
      if fringe is empty then return failure
      node ← REMOVE-FRONT(fringe)
      if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
      if STATE[node] is not in closed then
         add STATE[node] to closed
         fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

GRAPH-SEARCH (pg. 83) is an adaptation of TREE-SEARCH to handle cases in which multiple paths can lead to the same state. Here we create a *closed* list to explicitly store all the nodes that we have expanded and an *open* list to store all nodes on the fringe that are currently unexpanded.

When a previously visited state is removed from the fringe it will be tested in the if STATE[node] line and it will not be re-expanded. Note that this check is done after the state is removed and tested for the goal condition. It does not prevent repetitive states from being reinserted in the fringe. Also note that the algorithm will return the first goal solution (even when there are multiple paths to the same goal state of different costs) so can only be optimal if the first path encountered to a goal state is the cheapest.

To think about: the goal-test is performed before checking whether the state has been seen before. What ramifications does that have in search performance? Does it have any benefits?

Greedy best-first vs. A* search

Greedy best first expands the node that gives the least estimated cost to the goal: $f(n) = h(n)$. It ignores the step cost entirely.

A* combines uniform cost search and greedy best first search $f(n) = g(n) + h(n)$.

Figure 4.2 (pg. 96) shows GREEDY-BEST-FIRST-SEARCH finding a solution without backtracking.

To think about: is this always the case?

Also: Can you come up with search trees in which GREEDY-BEST-FIRST-SEARCH would be better suited than UNIFORM-COST-SEARCH? How about the other way around?

Note that in all cases that the appearance of a goal state in the fringe does not mean that the search will terminate right away and choose to expand that state. To think about: can you make any general statements that quantify the time in which a goal state appears on the fringe and when the search algorithm terminates with a goal?

Hill-climbing search

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
   inputs: problem, a problem
   local variables: current, a node
                    neighbor, a node

   current ← MAKE-NODE(INITIAL-STATE[problem])
   loop do
        neighbor ← a highest-valued successor of current
        if VALUE[neighbor] ≤ VALUE[current] then return STATE[current]
        current ← neighbor
```

HILL-CLIMBING is the first local search algorithm which we covered. It requires a complete state specification and moves between leaf nodes in the search tree by computing valid moves that end up in other complete states. It terminates at the maximal value of the objective function for some states.

Simulated annealing search

```
function SIMULATED-ANNEALING( problem, schedule) returns a solution state
   inputs: problem, a problem
           schedule, a mapping from time to "temperature"
   local variables: current, a node
                    next, a node
                    T, a "temperature" controlling prob. of downward steps

   current ← MAKE-NODE(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 current ← next
       else current ← next only with probability e^(ΔE/T)
```

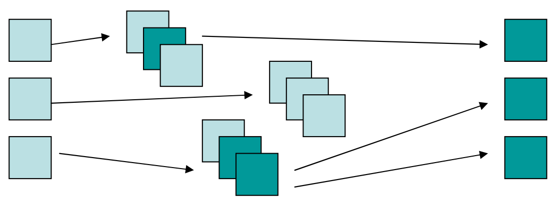
SIMULATED-ANNEALING allows the agent to take steps with decreases in the objective function with some probability. The probability depends on the schedule AND on the degree of "badness" of the move. In this sense all bad moves are not created equal. As more steps are taken in the algorithm, the probability of a backward move decreases and converges to only taking moves with increasing utility. If the rate of annealing is correctly set, the algorithm is guaranteed to find a solution if one exists.

Note that it picks a move at random. The probability transition is only used for moves with negative value. Any positive valued move (which increases the objective function's value) is taken with probability 1 if it is chosen, no matter how big or small the gain is.

To think about: what types of problems would this algorithm work well on? Work poorly on?

Local Beam Search

- Idea: instead of one state, keep track of many.
- Begins at k random states
- Generates all successors, keeps k best for next step.



LOCAL-BEAM-SEARCH can be seen as a cross between running k HILL-CLIMBING searches at random states and GENETIC-ALGORITHMS.

To think about: why is this?

Genetic Algorithms

(pg 119, figure 4.17)

- Consists three parts:
  - a pool of states (also called *individuals*)
  - genetic crossbreeding of states according to some fitness
  - mutation of population

To think about: what happens in the case when you remove one of these three parts from the genetic algorithm blueprint?

Minimax algorithm

```
function MINIMAX-DECISION(state) returns an action
   v ← MAX-VALUE(state)
   return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← −∞
   for a, s in SUCCESSORS(state) do
      v ← MAX(v, MIN-VALUE(s))
   return v

function MIN-VALUE(state) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← ∞
   for a, s in SUCCESSORS(state) do
      v ← MIN(v, MAX-VALUE(s))
   return v
```

MINIMAX-DECISION calculates a best move in a ply (two turns by opposing players). It assumes that it is moving for the first player, and thus wants to maximize its utility value for its move.

In odd levels of the tree the MAX-VALUE function is run, and at even levels, the MIN-VALUE function is run.

Calculation in the tree runs in a depth first manner until we reach a leaf (either at an even or an odd level) and the values are then propagated back up successively higher levels of the tree.

The either tree (initial state to the leaf nodes) must be searched before MINIMAX returns a decision. As this is not a realistic possibility (except for trivially small games) we must change this initial version of the algorithm such that decision can be made in reasonable (e.g. real) time.

The α-β algorithm

```
function ALPHA-BETA-SEARCH(state) returns an action
   inputs: state, current state in game
   v ← MAX-VALUE(state, −∞, +∞)
   return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
   inputs: state, current state in game
           α, the value of the best alternative for MAX along the path to state
           β, the value of the best alternative for MIN along the path to state
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← −∞
   for a, s in SUCCESSORS(state) do
      v ← MAX(v, MIN-VALUE(s, α, β))
      if v ≥ β then return v
      α ← MAX(α, v)
   return v
```

ALPHA-BETA-SEARCH adds alpha (max) and beta (min) values to the MINIMAX algorithm. In a node at a MAX level (an odd level), we know that its parent, a MIN node, will never choose this MAX node if there is another previously evaluated node with a known lower utility value. This is represented by beta (the value of the lowest-value choice along the path to that MIN node). In this case the search can be pruned and we can save execution time by not evaluating the remaining leaves of the tree.

In a node at the MIN level (an even level), we have a similar situation that uses alpha to decide whether to prune the node or not. Figure 6.5d (pg. 168) shows where a pruning action in the MIN-VALUE function kicks in. In Figure 6.5, pruning in MAX-VALUE doesn't occur. The pruning is brought about by the **return** statement in the SUCCESSORS function in the MIN- and MAX-VALUE procedures.