Comp3620/Comp6320 Artificial Intelligence
Tutorial 1: Search Formulations, Strategies, and Algorithms

March 13-16, 2018

Exercise 1 (problem formulation)

For each of the following problem, explain how states and actions can be represented, and give the initial state, goal test, successor function, and a plausible step cost function. Remember from the lectures that these elements constitutes a search problem formulation.

1. **Color a planar map using a minimum of colors in such a way that no two adjacent regions have the same color.**

Solution

We are given the number n of regions and an adjacency matrix $A[i,j], i,j \in 1,\dots,n$, where $A[i,j]$ is true iff region i is adjacent to region j. A state is a vector C, where $C[i], i \in 1,\dots,n$ is the color of region i or is NONE, and satisfies $A[i,j] \rightarrow C[i] \neq C[j]$ (or $C[i]$ is NONE).

In the initial state $C[i] = \text{NONE}$ for all i.

The successor function maps a state to a set of pairs, each consisting of an action applicable and a successor state. Formally, this is written as follows:

$$\text{successorFn}(C) = \begin{array}{r} \{\langle \text{color}(i,c), C' \rangle \mid C[i] = NONE \text{ and } C[j] \neq c \text{ for all } j \text{ such that } A[i,j] \\ \text{and } C'[i] = c \text{ and } C'[j] = C[j] \text{ for } j \neq i\} \end{array}$$

The step-cost function adds a cost of one for the use of a new color, and uses a small positive cost otherwise: $g(C, \text{color}(i,c), C') = 1$ iff $C[j] \neq c$ for all j and $= \epsilon$ otherwise.

The goal test is $goal(C)$ iff $C[i] \neq NONE$ for all i.

Alternative Solution

For those who don't like formal notations, a less formal, but still acceptable definition of the successor function would be as follows. The actions are $\text{color}(i,c)$ for coloring a yet uncolored region i with color c. For such an action to be applicable in state C:

- region i must be yet uncollored: $C[i] = \text{NONE}$
- color c must not have been used to color an adjacent region: $\forall j$ such that $A[i,j]$, $C[j] \neq c$

The resulting state C' is such that:

- C' is the same as C except that i has been colored with color c: $C'[i] = c$ and $C'[j] = C[j] \forall j \neq i$

2. **Using a water tap and three jugs of possibly different capacities (e.g. 12, 5, and 8 litres), collect a given amount of water (e.g. 9 litres) in one of the jugs as fast as possible. The available operations are: filling up a jug completely with the water tap, pouring the contents of one jug into another until one of them is full or empty, or completely emptying the contents of one jug onto the ground. Assume that moving 1 liter takes 1 unit of time.**

Solution

A state is a vector $S[i], i = 1 \ldots 3$, representing the current amount of liquid in the respective jugs. In addition, we know the (fixed) jug capacities $\mathrm{M}[i], i = 1 \ldots 3$ and the amount x to collect in one of the jugs.

The successor function is:

$$\text{successorFn}(S) = \begin{array}{l} \{\langle \text{empty}(i), S' \rangle \mid S'[i] = 0 \text{ and } S'[j] = S[j] \text{ for } j \neq i, i = 1 \ldots 3\} \cup \\ \{\langle \text{fill}(i), S' \rangle \mid S'[i] = \mathrm{M}[i] \text{ and } S'[j] = S[j] \text{ for } j \neq i, i = 1 \ldots 3\} \cup \\ \{\langle \text{pour}(i,j), S' \rangle \mid S'[j] = \min(\mathrm{M}[j], S[j] + S[i]), S'[i] = S[i] - S'[j] + S[j], \\ \qquad \text{and } S'[k] = S[k] \text{ for } k \neq i \text{ and } k \neq j, i, j = 1 \ldots 3, i \neq j\} \end{array}$$

The goal test is $\text{goal}(S) \equiv \exists i \in 1 \ldots 3 \text{ s.t. } S[i] = x$

The step-cost function can be equal to the amount of water moved at the rate of 1l per second:

- $\text{step-cost}(S, \text{empty}(i), S') = S[i]$,
- $\text{step-cost}(S, \text{fill}(i), S') = \mathrm{M}[i] - S[i]$,
- $\text{step-cost}(S, \text{pour}(i,j), S') = S[i] - S'[i]$.

Alternative Solution

Again, a less formal definition of the successor function, e.g. for the poor actioon, would be as follows. The actions are:

(a) $\text{pour}(i, j)$ for pouring the contents of jug i into jug j, with $i \neq j, i, j = 1 \ldots 3$. This action is always applicable in any state S, although if i is empty or j is full, it doesn't have much effect. The resulting state is S' such that:
   - jug j now additionally contains the full contents of jug i if that doesn't lead to exceed j's capacity, or else is completely full: $S'[j] = \min(\mathrm{M}[j], S[j] + S[i])$
   - jug i is left with whatever could not fit in j: $S'[i] = S[i] - S'[j] + S[j]$
   - any other jugs k is unchanged: $S'[j] = S[j]$, $\forall k \neq i$ and $k \neq j$

(b) $\text{empty}(i) \ldots$

(c) $\text{fill}(i) \ldots$

Side Remark

As an aside: For the example with 12, 8, 5 liter jugs, we can get 9 liters in 30 secs as follows: fill 12 liter jugg, pour 12 liter jug into 8 liter jug, fill 5 liter jug, pour 5 liter jug into in 12 liter jug. A more time consuming solution would be to fill the 8 liter jug, fill the 5 liter jug, poor the 8 liter jug into the 12 liter jug, pour the 5 liter jug into the 12 liter jug, empty the 12 liter jug, fill in the 8 liter jug, poor the 5 liter jug into the 12 liter jug, poor the 8 liter jug into the 12 liter jug.

3. **Three superheroes and three supervillains are on the side of a river, along with a boat which can hold one or two people. Find a way of getting them all to the other side, without ever leaving superheroes outnumbered by supervillains at any place. Naturally, the boat cannot move across the river by itself, and none of our super-humans can fly!**

Solution

The state S represents the number of superheroes ($\mathrm{h}(S) \in \mathbb{N}$) and supervillains ($\mathrm{v}(S) \in \mathbb{N}$) on the left, plus the position ($\mathrm{p}(S) \in \{\mathrm{l}, \mathrm{r}\}$) of the boat.

The initial state I is such that $\mathrm{h}(I) = \mathrm{v}(I) = 3$ and $\mathrm{p}(I) = \mathrm{l}$.

Actions move the boat plus 1 or 2 people from one side to another, subject to the numbering constraints (boat capacity constraints, no more people can be moved on the boat than exist on the side of the river they're being move from, no superhero left outnumbered on either side of the river). Hence the successor function is:

$$\mathrm{successorFn}(S) = \begin{array}{l} \{\langle \text{move-right}(v', h'), S'\rangle \mid \mathrm{h}(S') = \mathrm{h}(S) - h', \mathrm{v}(S') = \mathrm{v}(S) - v', \mathrm{p}(S) = \mathrm{l}, \mathrm{p}(S') = \mathrm{r} \\ \qquad 1 \le v' + h' \le 2, v' \le \mathrm{v(S)}, h' \le \mathrm{h}(S), \\ \qquad \mathrm{h}(S') < \mathrm{v}(S') \rightarrow h(S') = 0, \mathrm{v}(S') < \mathrm{h}(S') \rightarrow h(S') = 3\} \cup \\ \{\langle \text{move-left}(v', h'), S'\rangle \mid \mathrm{h}(S') = \mathrm{h}(S) + h', \mathrm{v}(S') = \mathrm{v}(S) + v', \mathrm{p}(S) = \mathrm{r}, \mathrm{p}(S') = \mathrm{l} \\ \qquad 1 \le v' + h' \le 2, v' \le 3 - \mathrm{v(S)}, h' \le 3 - \mathrm{h}(S), \\ \qquad \mathrm{h}(S') < \mathrm{v}(S') \rightarrow h(S') = 0, \mathrm{v}(S') < \mathrm{h}(S') \rightarrow h(S') = 3\} \end{array}$$

The goal is to have no-one on the left: $goal(S) \equiv v(S) + h(S) = 0$

The step-cost: 1 per action.

Alternative Solution

A less formal definition of the successor function would be:

(a) The actions that can be applied in a state $S = (h, v, \mathrm{l})$ are move-right(v', h') to move v' supervillains and h' superheroes to the right hand side of the border.

For such an action to be applicable, v, v', h, h' must:

- satisfy the fact that you can't put more people on the boat than you have on the left hand side of the river $v' \le v$ and $h' \le h$
- satisfy the boat capacity constraints: $1 \le v' + h' \le 2$
- leave no superhero outnumbered on the left hand side of the river: if $h - h' \le v - v'$ then $h - h' = 0$
- leave no superhero outnumbered on the right hand side of the river: if $3 - h + h' \le 3 - v + v'$ then $3 - h + h' = 0$

The action leads to the state S' with

- $h - h'$ super-heroes on the left hand side
- $v - v'$ super-villains on the left hand side
- boat position $p = \mathrm{r}$

(b) actions that can be applied in a state $S = (h, v, \mathrm{r})$ are similar.

Side Remark

As an aside, here is solution sequence:

state(v,h,p)	*action*	*comment*
$(3,3,\mathrm{l})$	move-right$(2,0)$	move 2 supervillains
$(1,3,\mathrm{r})$	move-left$(1,0)$	move 1 supervillain back
$(2,3,\mathrm{l})$	move-right$(2,0)$	move 2 supervillains
$(0,3,\mathrm{r})$	move-left$(1,0)$	move 1 supervillain back
$(1,3,\mathrm{l})$	move-right$(0,2)$	move 2 superheroes
$(1,1,\mathrm{r})$	move-left$(1,1)$	move 1 supervillain & 1 superhero back
$(2,2,\mathrm{l})$	move-right$(0,2)$	move 2 superheroes
$(2,0,\mathrm{r})$	move-left$(1,0)$	move 1 supervillain back
$(3,0,\mathrm{l})$	move-right$(2,0)$	move 2 supervillains
$(1,0,\mathrm{r})$	move-left$(1,0)$	move 1 supervillain back
$(2,0,\mathrm{l})$	move-right$(2,0)$	move 2 supervillains
$(0,0,\mathrm{r})$	goal!	

Exercise 2 (properties of search strategies)

True or False?

True [✓ False]: Depth-first graph search is guaranteed to return a shortest solution.

False: Depth-first provides no optimality guarantee at all. (It is only complete in finite search spaces).

[✓ True] False: Breadth-first graph search is guaranteed to return a shortest solution.

True. This is because frontier nodes are ranked by increasing order of depth. (The strategy expands all nodes whose depth is less than the depth of the shallowest goal node).

[✓ True] False: Uniform-cost graph search is guaranteed to return an optimal solution.

True: This is because frontier nodes are ranked by increasing order of $g(n)$. (The strategy expands all nodes such that $g(n) \leq C^*$, and some nodes such that $g(n) = C^*$ including one single goal node.)

True [✓ False]: Greedy graph search is guaranteed to return an optimal solution.

False: This is because it completely ignores the cost to reach frontier nodes from the initial state, and only looks into the future estimated cost to the goal.

[✓ True] False: Breadth-first graph search is a special case of uniform-cost search.

True: This is because breadth-first search behaves like uniform-cost search when all step costs are identical.

[✓ True] False: A* graph search with an admissible heuristic is guaranteed to return an optimal solution.

True: Note that this is only true of the version with node re-expansion. (Consistency is a stronger property than admissibility which does not require node re-expansion.)

True [✓ False]: A* graph search is guaranteed to expand no more nodes than depth-first graph search.

False: In dense solution spaces, depth-first could find a solution much faster than A*.

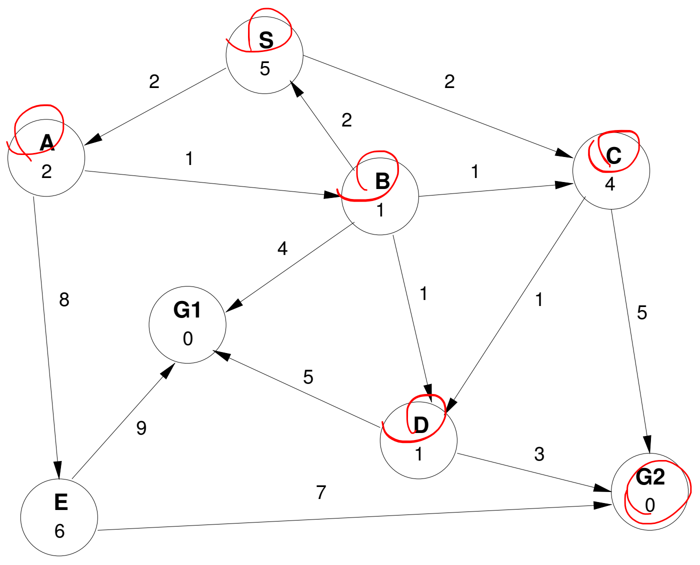
[✓ True] False: A* graph search with a consistent heuristic is guaranteed to expand no more nodes than uniform-cost graph search.

True: this is because uniform-cost amounts to A* with $h = 0$ which is the most uninformed heuristic. Uniform cost completely ignores the future cost to reach a goal. An alternative way to see this is to observe that uniform cost expand all nodes such that $g < C^*$ whilst A* only expand all nodes such that $g < C^* - h$ (with $h \geq 0$).

Exercise 3 (search strategies at work)

Consider the search space below, where S is the initial state and $G1$ and $G2$ both satisfy the goal test. Arcs are labelled with the cost of traversing them and the estimated cost to a goal is reported inside nodes.

For each of the following search strategies: breadth-first, depth-first, iterative deepening, uniform cost, greedy search, and A*, indicate the path found (if any) by graph-search and list, in order, all the states expanded — recall that a state is expanded when it is removed from the frontier. Everything else being equal, nodes should be removed from the frontier in alphabetical order. Assume that regardless of the strategy, the goal-test is performed when a node is dequeued from the fontier. For breadth-first and depth-first search, assume that newly generated nodes are not added to the frontier if they have already been explored or are on the frontier.



Breadth-first: finds S → C → G2. Expanded nodes: S(0), A(1), C(1), B(2), D(2), E(2), G2(2)

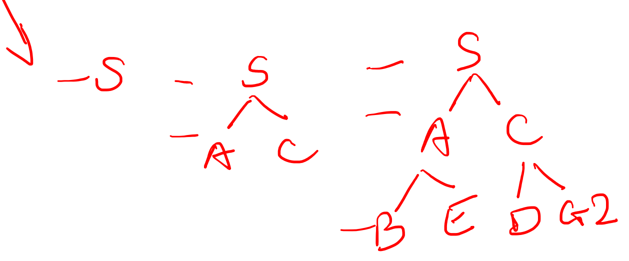
Depth-first: finds S → A → B → D → G2. Expanded nodes: S(0), A(1), B(2), D(3), G2(4)

? **Iterative deepending:** finds S → C → G2. Expanded nodes: S(0), S(0), A(1), C(1), S(0), A(1), B(2), E(2), C(1), D(2), G2(2)

Uniform cost: finds S → C → D → G2. Expanded nodes: S(0), A(2), C(2), B(3), D(3), G2(6)

Greedy: finds S → A → B → G1. Expanded nodes: S(5), A(2), B(1), G1(0)

A*: finds S → C → D → G2. Expanded nodes: S(0+5=5), A(2+2=4), B(3+1=4), D(4+1=5), C(2+4=6) D(3+1=4) G2(6+0=6)



Exercise 4 (Graph Search algorithm)

List all the issues in the following implementation of Graph Search, and explain their impact on completeness and optimality of the various strategies. Assume the EXPAND function is correct and as given in the lectures.

```
function GRAPH-SEARCH( problem, frontier) returns a solution, or failure
    explored ← an empty set of nodes
    frontier ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), frontier)
    loop do
        if frontier is empty then return failure
        node ← REMOVE-FRONT(frontier)
        if GOAL-TEST(problem, STATE[node]) then return node
        frontier ← INSERTNODES(EXPAND(node, problem), frontier)

function INSERTNODES( nodes, frontier) returns updated frontier
    for each n in nodes do
        add n to frontier
      if ∃ m in explored ∪ frontier s.t. STATE[m] = STATE[n] then
        PATH-COST[m] ← PATH-COST[n]
        PARENT[m] ← PARENT[n]
        ACTION[m] ← ACTION[n]
        DEPTH[m] ← DEPTH[n]
    return frontier
```

There are four issues with the above implementation.

1. In the Graph-Search function, nodes that do not match the goal-test should be added to the explored list. Otherwise, nodes labelled by the same state will be expanded multiple times, as in tree-search. For strategies such as depth-first search or greedy search, this can lead to incompleteness when there are loops in the search space.

   Loop,

2. One of the main principles of Graph Search is that at most one node labelled by a given state is on the frontier or the explored list at any one time. At the beginning of the InsertNodes function, successor node n should not systematically be added to the frontier. A node n should only be added at that point if no node m on the frontier or in the explored list is labelled by the same state as n. Otherwise we could compromise both optimality and completeness.

3. When a node m, is found on the frontier or in the explored list that is labelled by the same state as n, the various fields of m should only be updated if n represents a better path to that state, that is if $g(n) < g(m)$. Otherwise again optimality is compromised with strategies such as A* and even with uniform cost search.

4. Even if the above problems are fixed, the algorithm will only return the optimal solution if the heuristic is consistent. If the heuristic is only admissible then a node on the explored list should be re-opened when a better path to the state labelling that node is found. With a consistent heuristic, that case cannot happen because the first path to a state that is found (and which is put on the explored list) is the shortest path to that state.

Altogether, the correct version is:

```
function GRAPH-SEARCH( problem, frontier) returns a solution, or failure
    explored ← an empty set of nodes
    frontier ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), frontier)
    loop do
        if frontier is empty then return failure
        node ← REMOVE-FRONT(frontier)
        if GOAL-TEST(problem, STATE[node]) then return node
        add node to explored
        frontier ← INSERTNODES(EXPAND(node, problem), frontier)

function INSERTNODES( nodes, frontier) returns updated frontier
    for each n in nodes do
        ~~add n to frontier~~
      if $\nexists$ m in explored $\cup$ frontier s.t. STATE[m] = STATE[n] then
         add n to frontier
      else if PATH-COST[n] < PATH-COST[m]
        PATH-COST[m] ← PATH-COST[n]
        PARENT[m] ← PARENT[n]
        ACTION[m] ← ACTION[n]
        DEPTH[m] ← DEPTH[n]
        if m in explored then
            move m back to frontier
    return frontier
```

Recommended exercises from the book:

3.3, 3.6/b-c, 3.7, 3.9, 3.10, 3.11, 3.13, 3.14, 3.23