CS 188 Fall 2007 — Introduction to Artificial Intelligence — Section #1

Solutions

Problems to do in section

1. Uniform Cost vs. A* Search

a) Since we are performing graph search, we do not enqueue nodes where the state has already been expanded. Also, when putting nodes on the fringe, we do not check if a cheaper path to that state exists. This will create "dead nodes" on the fringe which will never be expanded since the state will have already been expanded and on the closed list when the cheapest path is popped off the fringe.

- **Fringe:** ([S],0)
  Pop: ([S],0)
- **Fringe:** ([S,A],2), ([S,E],3), ([S,B],5)
  Pop: ([S,A],2)
- **Fringe:** ([S,E],3), ([S,A,C],4), ([S,B],5), ([S,A,D],6)
  Pop: ([S,E],3)
- **Fringe:** ([S,A,C],4), ([S,B],5), ([S,A,D],6), ([S,E,B],7)
  Pop: ([S,A,C],4)
- **Fringe:** ([S,B],5), ([S,A,D],6), ([S,E,B],7), ([S,A,C,D], 7)
  Pop: ([S,B],5)
- **Fringe:** ([S,A,D],6), ([S,E,B],7), ([S,A,C,D], 7) ([S,E,B], 9), ([S,B,G],10)
  Pop: ([S,A,D],6)
- **Fringe:** ([S,E,B],7), ([S,A,C,D], 7), ([S,A,D,G],8) ([S,E,B], 9), ([S,B,G],10)
  Pop:([S,E,B],7) *Dead Node*
- **Fringe:** ([S,A,C,D], 7), ([S,A,D,G],8) ([S,E,B], 9), ([S,B,G],10)
  Pop:([S,A,C,D], 7), *Dead Node*
- **Fringe:** ([S,A,D,G],8) ([S,E,B], 9), ([S,B,G],10)
  Pop: ([S,A,D,G],8) *Found Goal*

b) We will represent fringe priority values as the sum $g(n) + h(n)$ for node n.

- **Fringe:** ([S],0)
  Pop: ([S],0)
- **Fringe:** ([S,A], 2+2),([S,E], 3+5),([S,B], 5+3)
  Pop: ([S,A],2+2)
- **Fringe:** ([S,A,C],4+1), ([S,A,D],6+1), ([S,E], 3+5), ([S,B], 5+3)
  Pop: ([S,A,C],4+1)

- **Fringe:**([S,A,D],6+1), ([S,E], 3+5), ([S,B], 5+3), ([S,A,C,D], 7+1)
  Pop: ([S,A,D],6+1)

- **Fringe:**([S,E], 3+5), ([S,B], 5+3), ([S,A,D,G],8+0) ([S,A,C,D], 7+1), ([S,A,D,B],7+3)
  Fringe: ([S,E], 3+5)

- **Fringe:** ([S,B], 5+3),([S,A,D,G],8+0) ([S,A,C,D], 7+1), ([S,A,D,B],7+3), ([S,E,B],7+3)
  Pop: ([S,B], 5+3)

- **Fringe:** ([S,A,D,G],8+0) ([S,A,C,D], 7+1), ([S,A,D,B],7+3), ([S,E,B],7+3), ([S,B,G],10+0), ([S,B,E],9+5),
  Pop: ([S,A,D,G],8+0) *Found Goal*

c) Since this is graph search, checking the heuristic is admissible is not sufficient. We must check that the heuristic is consistent. This entails checking that for each action a going from state n to n' satisfies:

$$c(n, a, n') \geq h(n) - h(n')$$

This amounts to checking that for edge in the search graph we have the cost of the edge is at least as big as the difference in heuristic values. For instance, for states E and B, the difference in heuristic values is 2 and the cost of the edge is 4. So consistency is satisfied for this edge.

2. Designing an A* Heuristic for Towers of Hanoi

a) Search State: First, assume that each of the n pegs are numbered in increasing order of size $1, \ldots, n$. We represent each peg as an ordered list of integers representing the discs on a peg. A search state is a three tuple (ℓ_1, ℓ_2, ℓ_3) of integer lists, where ℓ_i represents the ith peg. For instance, the starting state for $n = 5$ is $([1, 2, 3, 4, 5], [], [])$.

Successors and costs: We provide the python code for getting the successors and the cost of the action for each state.

```
def getSuccessors(state):
  successors = []
  for i in range(3):
      for j in range(3):
          if i != j:
              # Try move top disc from peg i to peg j
              if len(state[i]) > 0:
                  if len(state[j]) == 0 or state[i][0] < state[j][0]:
                      succ = [list(x) for x in state] # clone each list
                      disc = succ[i].pop(0) # top disc on peg i
                      succ[j].insert(0,disc) # put on top of peg j
                      successors.append( (succ,1) )
  return successors
```

Goal Test: Is the state $([], [], [1, 2, 3, 4, 5])$?

b) There are many correct answers. One possibility is n minus the number of discs on the third peg. Since the goal state requires all n discs on the third peg, we require at least that many moves to reach the goal. Therefore, our heuristic is admissible. Note that this heuristic corresponds to relaxing the constraint that smaller pegs rest atop larger ones.

Mini-Homework

Search

a) S→A→G

b) S→A→G

c) S→B→C→G

d) S→B→C→G

e) S→A→G

f) Node A

g) A search graph with a loop

h) A graph with a node with infinite branching factor

i) All costs in the search graph are 1.

Burrito Search Problem

a)

- **Search State**: Q_1, ..., Q_e are sets of task-time pairs, one for each employee, that describe what tasks s/he begins and when.
- **Actions**: $\texttt{Schedule}(T_i, Q_j, t)$ adds task T_i to employee js task set (Q_i) at time t.
- **Initial State**: $Q = \{\}$ for all i.
- **Goal Test**: Let R_i be the set of required tasks that must be completed before task T_i can begin. Then we reach a goal when all T are in some Q and begin after the completion of R.
- **Successor Function**: Let q_i be the earliest time that employee i can accept a new task (the end of its latest task). For tasks T_i with all of their required tasks R_i already scheduled, let r_i be the earliest time when all of those required tasks are completed. Then,

$$\texttt{Successor}(Q_1, ..., Q_e) = \{(\texttt{Schedule}(T_i, Q_j, t), Q_j = Q_j \cup (T_i, t) \text{ and } Q_k = Q_k \text{ for } k \neq j) \text{ for all } T_i \text{ with } R_i \text{ already in } Q_1, ..., Q_n \text{ and } t = \max(r_j, q_j)\}$$

- **Cost**: $\max(q_i)$ over all i. Note that q_i is the time at which employee i finishes his/her last scheduled task, as stated in the successor function description above.

b) A very simple heuristic comes from relaxing the constraint that tasks have dependencies. Let $t_{\max}$ be the length of the longest remaining unscheduled task, $T_{\max}$. Then, $h_1(Q_1, ..., Q_n) = \max(0, t_{\max} - max(q_i) \min(q_i)))$, the additional time required to complete $T_{\max}$ if it is delegated to the employee with the earliest available capacity.