Massachusetts Institute of Technology

16.410-13 Principles of Autonomy and Decision Making

Assignment #8, tutorial. Due: LEC #17

Problem PS.8.1.1: Best First from Depth First

The function below, although it is called `best-first` is really an an implementation of Depth-First search using the machinery in ps4/search.scm.

Your job is to convert that into a real definition for best-first search. For concreteness, add new (extended) paths onto the front of Q. You can assume that you have a working version of extend-path-with-heuristic. Also, note that we are assuming that the heuristic values already exist and you just need to look them up using `(get-heuristic-value node finish)`, as defined in the code.

```
define (best-first start finish)

  ;; Really, this is Depth-First!!



  ;; A Q addition function specific to DEPTH-FIRST search

  ;; Add the new paths to the front of the queue

  (define (merge-paths-into-Q new-paths Q)

    (Q-set-paths! Q (append new-paths (Q-paths Q)))

    )



  (define (heuristic node) #f)          ; not needed for DEPTH-FIRST



  (define (successors path)

    (extend-path path))
```

```
;; Fire up generalized search defined in search.scm:
(search
 ;; Just start with a partial path including only start node.
 finish                          ; goal node
 1                               ; only 1 path wanted
 pick-and-remove-first-path      ; pick the first path from Q
 merge-paths-into-Q              ; add to the front of Q
 successors                      ; successors of path
 ;; The initial Q, just one path = (start)
 (make-Q (make-path #f (list start))) ; initial Q
 '()                             ; initial wins
 ))
```

Problem PS.8.1.2: Robot Search

In PS 1 you saw that production rules, rules that add and delete assertions from the data base are quite sensitive to the order of firing. Simple-minded conflict-resolution strategies have a hard time ensuring that something useful happens. We saw, for example, that using simple rule-ordering, a set of rules aimed at modeling a simple robot just go into an infinite loop.

In this problem, we are going to exploit our new found expertise in searching to try to do better. In particular, we will use a best-first search to try to decide what rule-instance should be fired. You may be interested in how the forward chainer can be used to define an abstract graph to be searched, see ps8/robot.scm It is surprisingly easy.

To enable us to use heuristic search, you will need to provide a heuristic function to guide the search.

The basic idea for the heuristic function is as follows:

- The state of the world is described by a set of assertions. These are what we would find in the assertion database for our rule-based system.

- The goal state is decribed by another list of assertions, describing the final state. This does not have to be a complete description of the final state. The termination condition is when all the assertions in the goal state are present in the current state.
- The heuristic function should be a measure of the **difference** between these two states. The measure of difference will be specific to the particular problems we are dealing with, in particular, the types of assertions one finds in the states.

For variety, we will look at a different example. Let's look at rules for manipulating stacks of toy blocks. This is the "Blocks World" you have heard about in the book.

```
(B1 IF                              ; the table is always clear
    ADD      (clear table))

(B2 IF       (on ?x ?y)
             (clear ?x)
             (clear ?z)
     AND-IF  (not (eq? '?x '?z))
             (not (eq? '?y '?z))
     ADD     (on ?x ?z)
             (clear ?y)
     DELETE  (on ?x ?y)
             (clear ?z)
     SAYING  ("move " ?x " from " ?y " to " ?z))
```

There are only two kinds of assertions:

- `(on A B)` that indicates that A is ON B, and
- `(clear A)` indicating that A is clear, that is has nothing on it. The table is defined to be always clear, so you can always put something on the table.

Here are some assertions for a simple situation (state of the world):

```
(on A table) (clear A)
(on B table) (clear B)
(on C table) (clear C)
(on D table) (clear D)
```

The initial state (node) is precisely a list of these assertions. The goal state would be described with another list of assertions, which might not be a complete description of the world. For example:

```
(on A B)
(on B C)
```

is a perfectly reasonable description of a goal even though it does not specify where D is and what blocks are clear. Note that a path is simply a list of nodes and so it will be a list of lists of assertions.

Your job is to define the function `(get-heuristic-value state goal-state)` which is given two lists of (`on` and `clear`) assertions and returns an integer. The number should be low for states that might be "near" the goal-state, that is, which may require fewer rule applications to reach the goal.

Your code will be checked by testing how effective it is at reducing the number of search steps required to reach some sample goals. It will be compared to the "trivial" solution, using a heuristic function that returns 0 always. Your objective is to do better than this. This means that, in this case, our official answer (just return 0) does not qualify as correct. When after checking your solution, the system reports "the correct value is 14", this is simply what the system using a heuristic of 0 does. If your result is LESS than 14, it will be marked correct.

We will post some good solutions from students after the PS is due. Note, there is a time-limit for each problem evaluation, if your code exceeds that time limit, you will see an error message wondering if there is an infinite loop.

```
(define (get-heuristic-value state goal-state)

  your_code_here)
```

Problem PS.8.2.1: Heuristic Searches

In this question, we explore the key properties of some common heuristic searchs: hill-climbing (with and without backup), best-first and beam searches.

Part 1: Characteristics

Indicate which of the following statements are true.

1. Hill-climbing search (without backup) is guaranteed to find a path between the start and the goal if one exists (assuming finite search spaces).
2. Hill-climbing search (with backup) is guaranteed to find a path between the start and the goal if one exists (assuming finite search spaces).
3. Best-first search is guaranteed to find a path between the start and the goal if one exists (assuming finite search spaces).
4. Beam search (with beam-width less than d) is guaranteed to find a path between the start and the goal if one exists (assuming finite search spaces).
5. Beam search with beam-width of 1 is equivalent to hill-climbing without backup.
6. Hill-climbing search (with backup) and with a constant estimate of goodness for all nodes is equivalent to depth-first search.
7. The worst-case running time of best-first search is worse than that of hill-climbing search (with backup)
8. The space required by best-first search is worse than that required by hill-climbing search (with backup).
9. Hill-climbing search (with backup) is guaranteed to find the shortest path (measured in terms of number of nodes on the path) between the start and the goal.

10. Best-first search is guaranteed to find the shortest path (measured in terms of number of nodes on the path) between the start and the goal.
11. Best-first search is guaranteed to vist fewer nodes during the search than hill-climbing (with backup)

Part 2: Performance

In asnwering these questions assume that the search space is a tree with branching factor *b* and depth *d*. Note that the number of nodes in such a tree is $b^{(d+1)} - 1/ (b - 1)$. Since we only care about the order of growth, we will abbreviate this as $b^{(d+1)}$. The running time is proportional to the number of nodes visited and the space is proportional to the maximum length of Q in our simple implementation.

1. What is the (approximate) worst case running time for hill-climbing search (without backup)?
2. What is the (approximate) worst case space for hill-climbing search (without backup) ?
3. What is the (approximate) worst case running time for hill-climbing search (with backup)?
4. What is the (approximate) worst case space for hill-climbing search (with backup)?
5. What is the (approximate) worst case running time for best-first search?
6. What is the (approximate) worst case space for best-first search?
7. What is the (approximate) worst case running time for beam search (with beam-width = k)?
8. What is the (approximate) worst case space for beam search (with beam-width = k)?

Problem PS.8.2.2: A* Search

Indicate which of the following statements are true.

1. A* is guaranteed to find an optimal solution.
2. A solution found by A* is guaranteed to be optimal.
3. The heuristic in A* must never over-estimate the true cost of a solution.
4. A* makes use of an important principle from Dynamic Programming.
5. In a search of a square grid, allowed moves are North, South, East, and West. The cost of such a move is the distance traveled (1 mile). All nodes exist at intersections of the grid. The Euclidean distance (square root of vertical and horizontal distance) is not an appropriate A* heuristic.

Problem PS.8.3.1: Hours

We want to understand how much time it took students to answer the questions on the problem sets.

> Approximately how many hours did you spend really working on this problem set?