CS 510: Assignment 1

Fall 2012

Written

1. Does a finite state space always lead to a finite search tree? How about a finite state space that is a tree? Can you precisely state what types of state spaces always lead to finite search trees?

2. Consider "iterative lengthening search," an iterative analog to uniform cost search. The idea is to use increasing limits on path cost. If a node is generated whose path cost exceeds the current limit, it is immediately discarded. For each new iteration, the limit is set to the lowest path cost of any node discarded in the previous iteration. Is this algorithm optimal for general path costs? In other words, is it guaranteed to always produce the solution with the lowest possible path cost? Provide a formal proof either of optimality or the contrary.

3. What is the average branching factor of 8-puzzle[1]?

4. *Extra Credit:* What is the average number of "slides" required to solve an 8-puzzle?

Please show all work and cite all sources.

Programming

You will be implementing a solver for the 8-puzzle game. You will implement BFS, DFS, and A*.

For A^* you may use any heuristic (preferably admissible) of your choosing, however, simple Manhattan distance will suffice.

Once you have implemented all three search techniques, create a short report (just a couple paragraphs) comparing their efficiency. To be statistically sound, this will require running each algorithm a number of times over randomly-generated 8-puzzle instances. Either as a part of this report or a separate `README` should be instructions for running/testing your code.

The game logic has been implemented in `8puzzle.lisp` (available on the course website); you may choose to use this file as a starting point.

All work must be done in Lisp and be executable in `clisp` (which is installed on `tux.cs.drexel.edu`).

You should ultimately end up with a function called something like "`solve-8puzzle`" that, given an initial 8-puzzle instance will return a list of the moves required to get to the goal state. For example:

```
[1]> (load "8puzzle.lisp")
;; Loading file 8puzzle.lisp ...
;; Loaded file 8puzzle.lisp
T
[2]> (let ((puzzle (random-puzzle))) (print-puzzle puzzle) (solve-8puzzle 'BFS puzzle))
-------------
| 1 | 6 | 2 |
-------------
| 4 |   | 3 |
```

[1]For a description of 8-puzzle see `http://en.wikipedia.org/wiki/Fifteen_puzzle`

```
-------------
| 7 | 5 | 8 |
-------------
(UP RIGHT DOWN LEFT DOWN RIGHT)
```

Here are some hints/guidelines:

- Implement the General-Search algorithm described in R&N (this means you'll need the functions Make-Queue, Empty?, *et cetera*).
- Implement the queuing functions for BFS, DFS, and A*.
- You will probably need a function to determine whether the goal state has been reached and a function that generates successor states.
- The search-space of 8-puzzle is cyclic; you will probably need some way of making sure equivalent states are not expanded multiple times.
- Be aware that these algorithms take a long time to run. Depending on the speed of your processor and the randomly-generated 8-puzzle instance, DFS may take an inordinate amount of time to terminate. You may wish to read about configuring the Lisp compiler (compiled Lisp code will run orders of magnitude faster than interpreted code).