Assignment 1 - Solution

Ariel Stolerman

October 14, 2012

1

A finite state space does not always lead to a finite search tree. For instance, the state space in the Romania cities map example from the book is finite: {*In(Arad), In(Zerind),...,In(Eforie)*} (contains 20 states), however with the action *Go(<CityName>)* we can go through any circular path infinitely, which results with an infinite search tree.
In the case where the finite state space is a tree, there cannot exist circular paths (loops) by definition, therefore the search tree will be finite.
In fact, any legitimate structure that contains no circular paths with a finite number of states will also have a finite search tree. Finite directed acyclic graphs (DAGs) fit the description, as by definition they contain no loops (and directed simply means the actions that map from one state to another are not symmetric, so you cannot jump between two adjacent states infinitely).

2

The algorithm is optimal since nodes selected to be expanded are extracted from the fringe by increasing path costs, therefore the optimal goal will be reached before any other goal. Following is a formal proof:
Let there be a search tree with an initial node s, an optimal goal G_1 and a suboptimal goal G_2. In addition let n' be the lowest common ancestor of G_1, G_2 and n the first immediate successor of n' on the path to G_1.
If $n = G_1$, it could not have been that G_2 is selected, since G_1 is optimal and so by definition it has the lowest path cost and would have been selected.

In any other case, since the path to G_2 goes through n', n must have been added to the fringe. Since G_2 has been selected, it must be that

$$g(G_2) \leq g(n) \tag{2.1}$$

But since G_1 is optimal and G_2 is suboptimal, and all path costs are non decreasing, then

$$g(n) < g(n) + \epsilon \leq g(G_1) < g(G_2) \tag{2.2}$$

for some $\epsilon > 0$ (an action minimum cost as defined for standard uniform search). Therefore $g(n) < g(G_2)$, in contradiction to 2.1, and so G_1, the optimal goal, must have been selected. □

3

There are 3 types of states leading to 3 possible branching factors depending on the position of the blank:

- Middle of row / column adjacent to a wall: there are 4 such states, with a branching factor of 3 (move along the wall or to the center).
- Corner: there are 4 such states, with a branching factor of 2 (move along any of the 2 adjacent walls).
- Center: there's only one such state, with a branching factor of 4 (move up, down, left or right).

So the average branching factor is

$$\frac{(4 \times 3) + (4 \times 2) + (1 \times 4)}{4 + 4 + 1} = \frac{24}{9} = 2\frac{2}{3} \tag{3.1}$$

Note: I assume the probability of the blank being at any of the 9 possibilities is equal, in all states (therefore the weight of the branching factor of each blank position is the same in the average calculated above).

Code Assignment: 8-Puzzle

Running the Code

Note: To avoid stack-overflow, please compile the code before use.
The main search function to run the code is `graph-search-sol (puzzle add-func)`, where `puzzle` is the puzzle to solve and `add-func` is the function to be used to add nodes the fringe in the search, and should be one of:

- `add-bfs`: breadth-first search, the fringe is maintained in LIFO.
- `add-dfs`: depth-first search, the fringe is maintained in FIFO.

- `add-a*`: A*, the fringe is maintained as a min-priority queue w.r.t. the Manhattan distance heuristic.

The function `graph-search (puzzle add-func)` does the same, only returns a list where the first element is the solution sequence, the second is the maximum fringe size reached and the third is the maximum depth reached.
An example run:

```
> (let ((p (random-puzzle))) (print-puzzle p) (graph-search-sol p #'add-bfs))
-------------
| 2 | 4 | 1 |
-------------
| 5 |   | 3 |
-------------
| 7 | 8 | 6 |
-------------
(LEFT UP RIGHT RIGHT DOWN LEFT LEFT UP RIGHT DOWN RIGHT DOWN)
```

There are 3 more test functions to run batch-tests / check output solutions:

- `test (add-func &optional (max-moves 20) (num-runs 10)`: runs a batch of `num-runs` tests over randomly generated puzzles with `max-moves` and outputs a list of solution statistics: average running time (in milliseconds), minimum / maximum / average solution length in moves, and minimum / maximum / average fringe size reached. Example run:

  ```
  > (test #'add-a* 30 5)
  (18.20104 2 12 8.8 4 259 117.8)
  ```

- `test-all (&optional (max-moves 20) (num-runs 10)`: runs `test` with all 3 functions: `add-bfs, add-dfs, add-a*`.

- `test-solution (puzzle solution)`: checks whether the given solution sequence is correct for the given puzzle. Returns `T` or `NIL` accordingly.

Performance Report

Statistics for running 100 experiments per algorithm, with randomly generated puzzles limited to maximum 20 moves, are presented in table 3.1.

Algorithm	BFS	DFS	A*
Avg. Runtime (ms)	79.30	134,381.0	7.44
Min. Moves	2	2	2
Max. Moves	16	114,008	14
Avg. Moves	6.44	41,629.1	6.5
Min. Max. Fringe Size	6	6	6
Max. Max. Fringe Size	12,260	146,356	565
Avg. Max. Fringe Size	481.94	53,200.95	53.15

Table 3.1: Performance results

It can be seen that DFS performs by far worse than BFS or A* with Manhattan-distance heuristic (this difference may be narrowed by upgrading to iterative deepening DFS) in all measurements: running-time, solution lengths and fringe size. Out of the top two, it can be seen that A* outperforms BFS in terms of running-time and memory usage (by a factor of 10–20).