COMP3702/7702 Artificial Intelligence
Semester 2, 2019
Tutorial 2 - Sample Solutions

Question 1

State Space:
The 8-puzzle problem contains 9 cells stored in a grid. This can be represented as a string of 9 character, with an '_' where the empty space is:

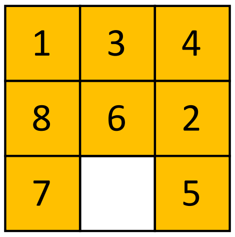


Figure 1: Example 8-puzzle state, which can be represented as the string 1348627_5

Action Space:
The action space only consists of 4 actions that are, move the blank space: {`up`, `down`, `left`, `right`}. Note that some actions are invalid, e.g. in the example above, moving the blank space `down` would be invalid, as there isn't anything there.

One way to identify invalid actions programmatically is by observing the row and column of the blank space. Using 1-indexing, the first element in the string will be at index 1.

- If the blank space is in the top row, then its index in the string is less than 4. In this case, an `up` action would be invalid. Similarly, if its index is greater than 6, then it is in the bottom row, and a `down` action would be invalid.
- `left` and `right` invalid moves can be determined by the modulus operator. If the blank index % 3 is 0, then it is in the right column, and a `right` action is invalid. If the index % 3 is equal to one, then it is in the left column, and the `left` action is invalid.

BFS and DFS:
There are a total of 9! states in the 8-puzzle problem (= 362,880). It would take a long time to store each one of them. Instead, they can be created as the search is performed.

A node in a tree can be represented as in Figure 2.

In BFS, the successors of a node get added to a First In First Out (FIFO) queue. This can be implemented in Python using `container.append(next_node)`, and `container.pop(0)`. The successors will be the the neighbours of the current state. For each state there can be a maximum of 4 neighbours.

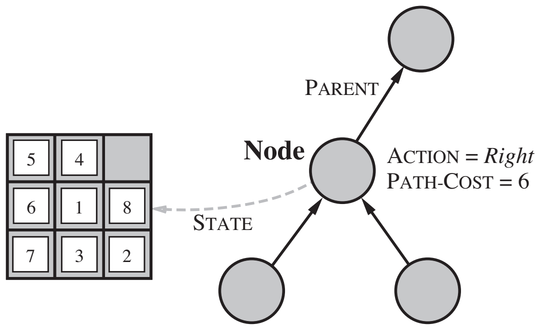
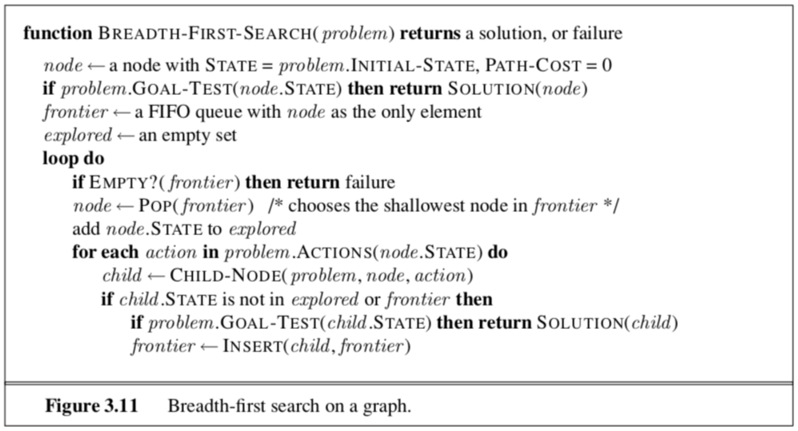


Figure 2: Example of a node which represents its STATE, stores its PARENT, the ACTION that was performed from the parent to arrive at the node, and the PATH-COST to arrive at that node (Source: Russell & Norvig textbook)

e.g. 1348627_5 has 3 neighbours: (1348_2765, 134862_75, 13486275_)

1348627_5 would be the first state to get constructed and added to the queue. It is checked whether it is the goal node, gets marked as expanded, and then its neighbours get constructed and added to the queue.

The pseudocode for BFS is shown in Figure 3



```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
      if EMPTY?(frontier) then return failure
      node ← POP(frontier)  /* chooses the shallowest node in frontier */
      add node.STATE to explored
      for each action in problem.ACTIONS(node.STATE) do
          child ← CHILD-NODE(problem, node, action)
          if child.STATE is not in explored or frontier then
              if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
              frontier ← INSERT(child, frontier)
```

Figure 3.11 Breadth-first search on a graph.

Figure 3: Pseudocode for Breadth-First-Search on a graph (Source: Russell & Norvig textbook)

An example implementation is attached in the zip file.

In DFS, the same procedure is run, but the successors of the node get added to a Last In First Out (LIFO) stack. A LIFO queue means that the most recently generated nodes are expanded first. This must be the deepest unexpanded node because it is one deeper than its parent – which, in turn, was the deepest unexpanded node when it was selected. This can be implemented in Python using `container.append(next_node)`, and `container.pop()`. An alternative implementation is to implement depth-first search with a recursive function that calls itself on each of its children in turn.

Question 2

Run your puzzle solver to generate the Sequence of Actions, Number of Steps, and Time taken.

Example values (with goal-check on generation):

- startState = 1348627_5; goalState = 1238_4765
  BFS: steps = 6, visited = 22, time = 0.006s;
  DFS: steps = 836, visited = 854, time = 0.0172s
- startState = 281_43765; goalState = 1238_4765
  BFS: steps = 10, visited = 219; time = 0.03s;
  DFS: steps = 69434, visited = 72441, time = 253.44s
- startState = 281463_75; goalState = 1238_4765
  BFS: steps = 13, visited = 1120; time = 0.266s
  DFS: steps = 82141, visited = 86490; time = 361.0s

The solution for all 3 cases is a relatively small distance away from the start node. BFS takes a very small number of steps, while DFS takes a very large number of steps. This can happen in cases where the solution is a small number of actions away, as BFS prioritises an optimal number of actions, while DFS is not guaranteed to be optimal, and expands its search to bigger depths and larger sequences.

BFS also requires significantly less time to solve the problem than DFS in each of the given cases. This result is in line with the time complexity of each algorithm – the time complexity of BFS has exponential dependence on d, the depth of the minimum depth goal, while the time complexity of DFS has exponential dependence on the m, the maximum depth of the search tree.

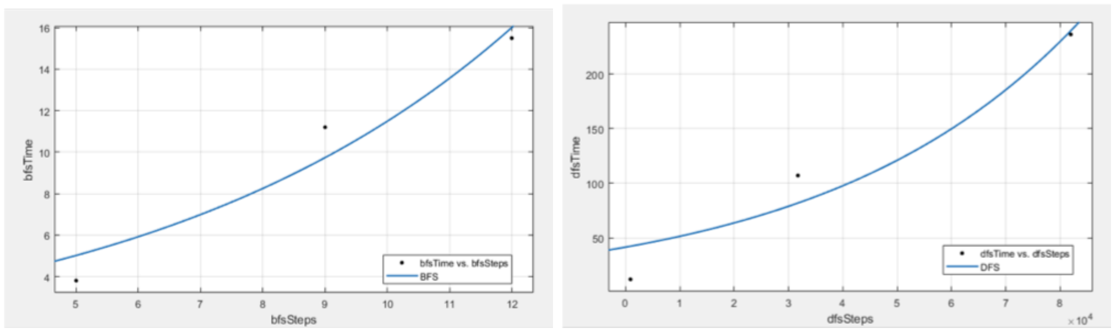


Figure 4: Time complexity for BFS vs DFS)

Recall time complexity for BFS is $O(b^d)$, and for DFS is $O(b^m)$. Time complexity can be calculated by performing regression on this data (try using Matlab's cftool). It is possible that the results may not fit the expected complexity for BFS and DFS. This can be the result of overhead in the calculations, or lack of datapoints – try adding more tests to the regression curve.

Space can be approximated by observing the maximum number of nodes in the queue for BFS or number of nodes in the stack for DFS at any time.

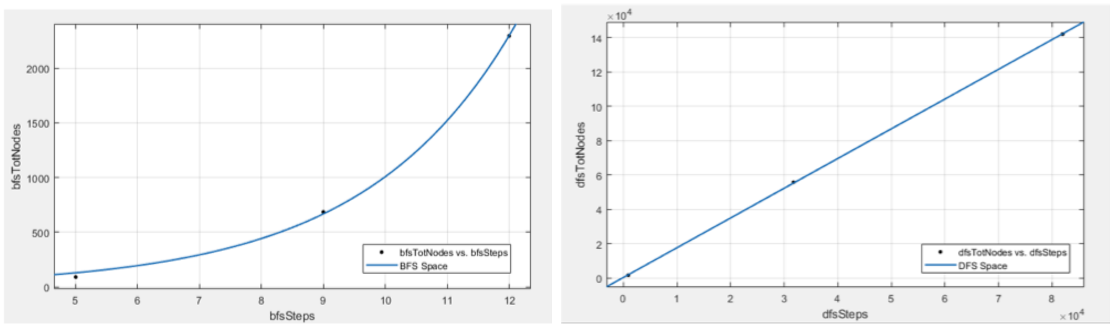


Figure 5: Space complexity for BFS vs DFS)

Space complexity for BFS is $O(b^d)$, the number of nodes expanded, and for DFS is $O(bm)$, for the 8-puzzle problem, the branching factor is 4. For DFS, the number of nodes in the stack is increased with the maximum depth of the tree – you can try changing the max depth in the code and observe the results. This is called depth-limited search. A combination of DFS and BFS can be implemented called Iterative-deepening depth first search (IDDFS), where we use depth-limited search with increasing search depth.

Question 3

Recall that BFS performs a complete search, given a finite branching factor. Thus, BFS will search all possible combinations of paths within the search space, and eventually run out. When the search has run out of paths to search it can conclude that there is no solution, thus BFS can output the right answer.

DFS can also perform a complete search if m and b are finite and states are not revisited. Although the path may not be optimal, DFS can also output the right result.

Parity

(See supporting notes)

Parity can be used to determine if the goal node is reachable from the start node, if the start and goal have the same parity, it is reachable, otherwise it is not. This can significantly improve the size of the state space, as half of the states can be ignored immediately, as they are unreachable. The state space for the 8-puzzle problem is 9!, that's a big number. Using parity checking, this can be reduced by a factor of 2.

Parity can be calculated by finding the number of inversions of a state mod 2.

- The number of inversions of tile-i ($N(s,i)$) is the number of tiles that appear after tile-i that have a smaller value than tile-i.
- The parity is the total sum of the inversions of each tile, mod 2
  Parity $= \sum_{i=1}^{n} N(s,i) \bmod 2$

Question 4

a) Yes

b) They should be identical[1]

[1] Note: in some implementations there might be a difference in when the goal test is applied. e.g. in BFS the goal test is usually applied to each node when it is generated rather than when it is selected for expansion. In Uniform-Cost-Search, the goal test is usually applied to a node when it is selected for expansion rather than when it is first generated. The reason is that the first goal node that is generated may be on a suboptimal path. The second difference to BFS is that a test is added in case a better path is found to a node currently on the frontier.