Documentation and Answers for the Implementation of the BFS and DFS Algorithms

Julianne Marie Ruz

February 17, 2020

1 Introduction

The BFSearch and DFSearch algorithms are both variations of the TreeSearch algorithm, differing only in how new nodes are added to their graphs. As part of the requirement for CMSC 170 - Artificial Intelligence, we were asked to document the results of our respective implementations of these two algorithms using the 8-puzzle game.

2 Implementation

2.1 Overview

In my implementation, I decided to create a separate State class to hold the required fields: the current puzzle layout, the list of moves needed to reach the current puzzle layout, and the state's parent node. I also decided to use ArrayLists to store the states. Certain parts of the algorithms I split into separate functions to better keep track of any errors I might encounter, as well as to keep the source code readable. These separate functions I shall explain below.

2.2 BFSearch

2.2.1 Documentation

Based on the pseudocode provided in the lab handout, we can see that the BFSearch algorithm accesses nodes in a first-in-first-out manner similar to a queue. To achieve this, I only removed the first element from the Arraylist.

```
int frontier_index = 0;
...
current_state = frontier.get(frontier_index);
frontier.remove(frontier_index);
```

Each state's possible moves were determined using a get_allowed_moves() getter function built into the State class that can access the results of a find_possible_moves() function that uses the state's current puzzle layout as a basis for finding the correct possible moves. In addition to that, the State class also uses a list_previous_moves() function to add the parent node's move path to the current node's move path.

In the BFSearch algorithm implementation, one can see the use of a compare_boards() function and a find_results() function. These two I programmed separate from the main algorithm because the way that I had programmed the State class meant that each state would have a different move path and parent node regardless of their current puzzle layout. Thus the compare_boards() function was created as a means to compare states based on their current puzzle layout. Puzzle layouts were stored in 2D arrays and were compared using for loops to check for discrepancies in the positions of values.

The find_results() function is the equivalent of the Results() function mentioned in the pseudocode provided in the lab handout and returns a new instance of the State class given a State and a move (indicated by a String). Additionally, the detect_win_state() function is the same as the GoalTest() function and compares a state's current puzzle layout to a predetermined winning puzzle layout provided by the make_win_condition() function.

2.2.2 Answers

Two test cases (test1.in and test2.in respectively) were provided in the lab class to test our implementations. The BFSearch function performed well on the first test case but took a substantial amount of time to try and provide a solution to the second test case and unfortunately concluded with an out-of-memory error.

2.2.3 Drawbacks of Implementation

I realize in hindsight that the use of the ArrayList class as opposed to the LinkedList class was one of the contributing factors to the BFSearch implementation's failure to find a solution to the second test case as runtimes for adding/deleting elements from the ArrayList are slower compared to the runtime for searching for an element. These adding/deleting runtimes are faster for other classes such as the LinkedList.

2.3 DFSearch

2.3.1 Documentation

The DFSearch algorithm is similar in structure to the BFSearch algorithm, so I decided to modify it to fit the requirements of the DFSearch algorithm instead of building a new one from scratch.

2.3.2 Answers

As with the BFSearch algorithm, the DFSearch algorithm did find the solution to the first test case though, as is in its nature of checking paths from root to leaf node instead of doubling back to check by row as the BFSearch algorithm does, it did take longer to come to the same conclusion. It was also susceptible to the out-of-memory error that hampered the BFSearch algorithm's performance, so I was forced to implement a cut-off for the number of nodes it checked so I could see whether or not it was performing as it should.

2.3.3 Drawbacks of Implementation

Since the two algorithms are similar except for how they store and access the states, the DFSearch could have also benefited more from the use of a different data structure that was more suited to its needs.

3 Conclusion

While both algorithms are useful, I would prefer to use the DFSearch algorithm to find a solution to the 8-puzzle game. Granted, the algorithm does take a long time to find the correct answer, but it can be supplemented with extra modifications in order to find certain patterns i.e. paths cycling through several states only to end up back where it started, present in the graph, resulting in a more thorough search for the answer.