Tile Game Report

Anna Nguyen and Ethan Youso

March 2022

1 Introduction

The 8-tile puzzle is a game in which numbered tiles are arranged in a 3x3 grid. To win the game, the user must configure the tiles into the desired order, preferably using the fewest number of moves possible. Search algorithms may be applied to this problem to find solutions. In the past, 8-puzzle has been used as a classical problem to analyze different search techniques [1]. In this project, breadth-first search, depth-limited search, and A* search were utilized to solve 8-tile puzzle. This report will compare the average number of states expanded, states left on the fringe, and maximum states stored of each search. Additionally, this report includes the approach to implement each algorithm, their test cases, and the results.

2 Approach

The Tile Game was written in Python 3.8.2. Python was chosen because it is a high-level language and has many useful packages. Additionally, the code of the node class was inspired by Professor Heather from the Artificial Intelligence class. All three algorithms were developed based on Russell's Artificial intelligence : a modern approach [2].

To implement Breadth-First Search (BFS), an array was used to store the explored states. Explored states are the nodes that are popped off the fringe. The fringe is made up of nodes that have not been explored yet. The memory is comprised of both explored states and the fringe. The array was chosen because it provides easy access to the elements, and the insertion and deletion can be done easily. As seen in listing 1, the breadth-first search will not stop until the fringe is empty. The BFS will pop the node out of the fringe using a first in first out (FIFO) method. If the current node is not the

goal, and the depth of the current node is less than 10, the BFS search will find the node's current children and add to the fringe and explored array.

Listing 1: Breadth-first search

```
def search(start, goal, searchType):
    explored = [] #states I have seen
    fringe = [] #stored kids
    fringe.append(start)
    explored.append(start.state)
    numberExplored = []
    cost = 0
    while fringe:
        # BFS
        if searchType == 1:
            node = fringe.pop(0)
        #calculate the cost for each node
        cost += node.calcCost()
        #increment the number of node explore
        numberExplored.append(node)
        if node.isGoal(goal):
            rebuildSolution(node)
            return numberExplored, explored, fringe
        else:
            if searchType == 1:
                kids, moves = node.findKids(1)
            for i in range(len(kids)):
                if(kids[i] not in explored and node.getDepth() <= 10):
                    newNode = Node(kids[i], node, moves[i], False)
                    fringe.append(newNode)
                    explored.append(kids[i])
    return numberExplored, explored, fringe
```

Similarly to BFS, Depth-limited search (DLS) also utilizes arrays to store the memory, fringe, and explored states. However, instead of utilizing a first in first out array, DLS uses a last in first out (LIFO) array. Specifically, the only two main parts that are different from BFS are included in listing 2. Since the search method uses LIFO, it also uses a different **findKids** method. Instead of adding to the kids array all the states in clock-wise order, DLS will add the states in the opposite order (up, left, down, right).

Listing 2: Part of depth-limited search

```
# DLS
    elif searchType == 2:
    node = fringe.pop()
    kids, moves = node.findKids(2)
```

To implement A* search, a priority queue was utilized. Specifically, Python's priority queue package was used to create the priority queue. The element with the lowest cost will have higher priority and will be dequeued first. The total cost of an element is the combined cost of g(n), the cost to reach the node, and h(n), the heuristic cost. The g(n) cost is the cost of the move, and the h(n) is calculated from the number of misplaced tiles. Since each step costs is 1, so the path cost g(n) is the number of steps. The g(n) code (the diffTiles method) is similar to BFS and DLS is shown in listing 3. The code to calculate the number of misplaced tiles is shown in listing 4.

Listing 3: Method to calculate the g(n) cost

```
def calcCost(self):
    """cost of this move - cumulative from parent"""
    if(self.depth == 0):
        return 0
    else:
        if self.action == "U":
            return self.getParent().getCost()+1
        elif self.action == "R":
            return self.getParent().getCost()+1
        elif self.action == "D":
            return self.getParent().getCost()+1
        elif self.action == "L":
            return self.getParent().getCost()+1
```

Listing 4: Method to calculate the h(n) cost

```
def diffTiles(self, goal):
    count = 0
    for i in range(len(self.state)):
        if self.state[i] != goal[i]:
            count += 1
    return count
```

Twenty tests were performed, with five different levels of difficulty. The levels are solvable in one move, two moves, four moves, eight moves, and ten moves. Each level had four tests. The results of the tests gave us the average

number of states expanded, the average number of states left on fringes, and the average of maximum states stored. The standard deviation of all four test cases were also calculated using the statistic package in Python. All test cases is included in the appendix.

3 Results

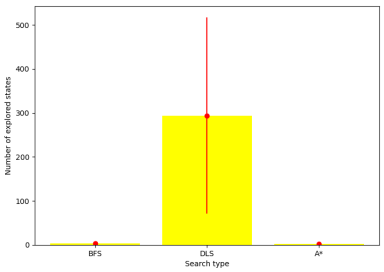
In theory, BFS would expand $O(b^d)$ with b as the branching factor and d as the depth of the shallowest solution. Assuming the branching factor is 3, BFS would expand a maximum 3^1, DLS would expand a maximum $3^{10} = 59049$, and A* would also expand a maximum 3^1 for experiments where the solution was found in 1 move. Based on **Figure 1**, BFS explored 3 nodes, DLS explored 294 nodes, and A* explored 2 nodes. BFS's results match the expected theoretical performance of the algorithms. DLS's number of explored nodes was less than the expected theoretical performance, and this might be because avoiding the repeating state would decrease the branching factor. Additionally, if the puzzle only needed one up move to solve, DLS would expand only one node. However, if the puzzle needed one left move, DLS would expand many nodes. This resulted in a large standard deviation. A* search also resulted in less than the expected theoretical performance, this might be because avoiding the repeating state also decreases the branching factor for A* search.

Moreover, DLS had the most nodes in memory. Since DLS had the most explored nodes, it was reasonable that DLS had the least number of nodes left on the fringe. Additionally, DLS had the largest standard deviation because it did not take the same number of explored nodes for every puzzle that could be solved in one move. Specifically, the puzzle that can win with one **up** move will take the least amount of explored nodes, but the puzzle that can be won with one **left** move will result in more explored nodes. A* search approach resulted in the smallest amount of explored nodes.

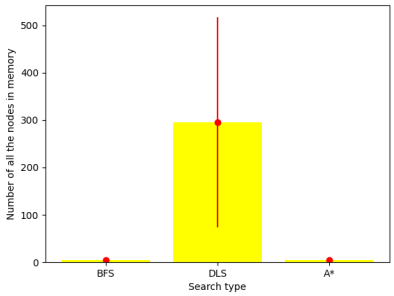
In the experiments where the solution was found in 2 moves, the results are mostly similar to the experiment where the solution was found in 1 move (**Figure 2**). The only difference was that the number of nodes left on the fringe for A* was less than BFS. This may be because as the number of moves required to win increases, the difference between the number of nodes explored by BFS and DLS also increases. This also results in the difference between the number of nodes on the fringe of BFS and DLS increasing. In the experiments where the solution was found in 4 moves, A* is still the best search. A* results in the smallest number of explored nodes, memory,

and nodes on the fringe (**Figure 3**).

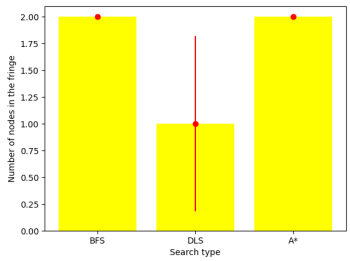
The difference in the performance of all three searches was observed in the last two tests, where the puzzle is solved in four moves and eight moves. A* is still the best search, which results in the smallest average number of explored nodes and the smallest average number of nodes in memory. Depth limited search is still the worst search approach since it had the largest average number of explored nodes and nodes in the memory. However, the standard deviation of BFS increased as the number of moves required to win increased. This might be due to the bias in picking the puzzle boards.



(a) The average number of explored nodes for BFS, DLS, and A* were 3.0 ± 0.8 , 294 ± 223, and 2 ± 0 respectively.

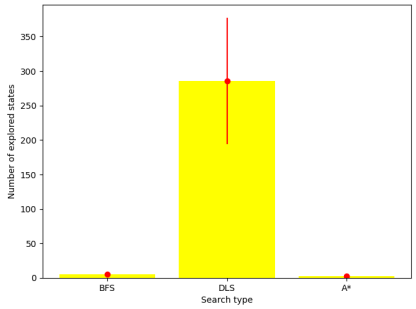


(b) The average number of nodes in memory for BFS, DLS, and A* were 5.0 ± 0.8 , 295 ± 222, and 4 ± 0 respectively

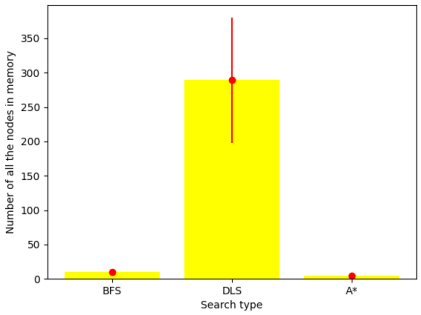


(c) The average number of nodes in fringe for BFS, DLS, and A* were 2 ± 0, 1 ± 0.8, and 2 ± 0 respectively

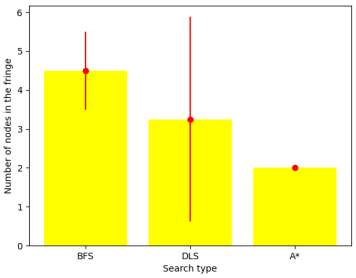
Figure 1: The average number of explored nodes, in memory, in the fringe for 3 search types (BFS, DLS, and A*) with error bar for 4 experiments where the solution can be found in 1 move



(a) The average number of explored nodes for BFS, DLS, and A* were 5.8 ± 0.5, 286 ± 91, and 3 ± 0 respectively

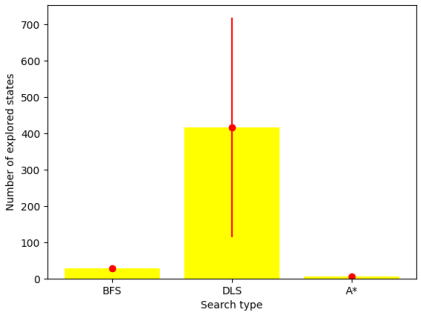


(b) The average number of nodes in memory for BFS, DLS, and A* were 10 ± 2, 289 ± 91, and 5 ± 0 respectively

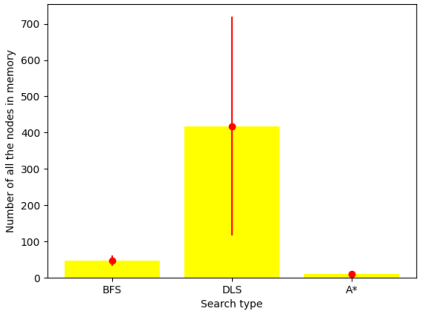


(c) The average number of nodes in fringe for BFS, DLS, and A* were 4.5 ± 1, 3.3 ± 2.6, and 2 ± 0 respectively

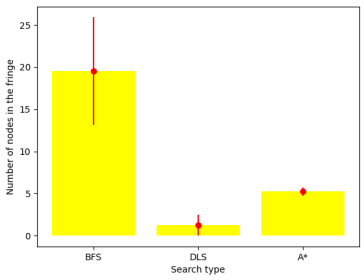
Figure 2: The average number of explored nodes, in memory, in the fringe for 3 search types (BFS, DLS, and A*) with error bar for 4 experiments where the solution can be found in 2 moves



(a) The average number of explored nodes for BFS, DLS, and A* were 28 ± 9, 417 ± 302, and 6 ± 0 respectively

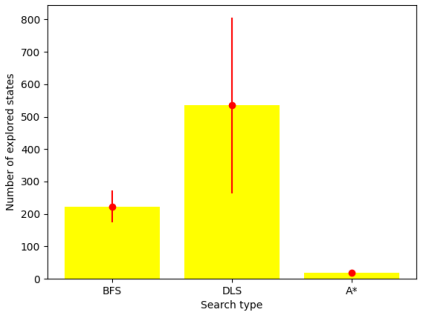


(b) The average number of nodes in memory for BFS, DLS, and A* were 48 ± 14, 418 ± 301, and 11.3 ± 0.5 respectively

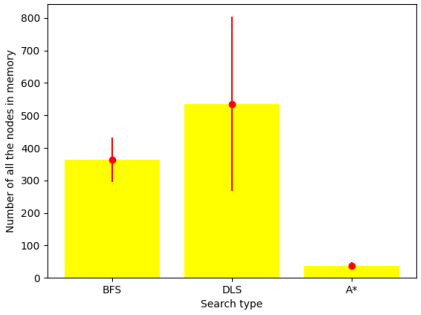


(c) The average number of nodes in fringe for BFS, DLS, and A* were 20 ± 6, 1 ± 1, and 5.3 ± 0.5 respectively

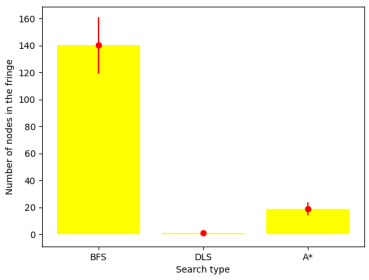
Figure 3: The average number of explored nodes, in memory, in the fringe for 3 search types (BFS, DLS, and A*) with error bar for 4 experiments where the solution can be found in 4 moves



(a) The average number of explored nodes for BFS, DLS, and A* were 223 ± 49, 534 ± 270, and 19 ± 5 respectively

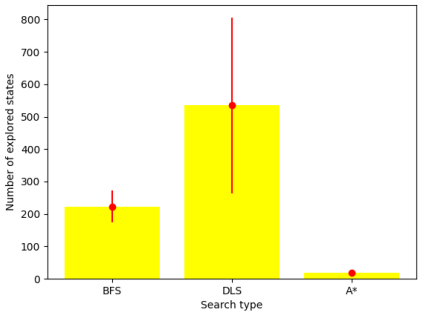


(b) The average number of nodes in memory for BFS, DLS, and A* were 363 ± 68, 536 ± 268, and 38 ± 10 respectively

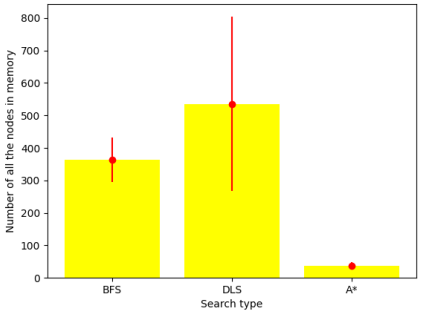


(c) The average number of nodes in fringe for BFS, DLS, and A* were 140 ± 21, 1 ± 2, and 19 ± 5 respectively

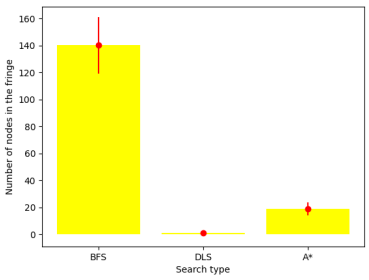
Figure 4: The average number of explored nodes, in memory, in the fringe for 3 search types (BFS, DLS, and A*) with error bar for 4 experiments where the solution can be found in 8 moves



(a) The average number of explored nodes for BFS, DLS, and A* were 617 ± 134, 481 ± 278, and 39 ± 8 respectively



(b) The average number of nodes in memory for BFS, DLS, and A* were 987 ± 199, 483 ± 276, and 75 ± 12 respectively



(c) The average number of nodes in fringe for BFS, DLS, and A* were 370 ± 66, 2 ± 2, and 35 ± 6 respectively

Figure 5: The average number of explored nodes, in memory, in the fringe for 3 search types (BFS, DLS, and A*) with error bar for 4 experiments where the solution can be found in 10 moves

4 Conclusion

In this report, we present the implementation of three different kinds of search: breadth-first, depth-limited, and A*. After performing 20 tests and comparing the number of nodes explored, in memory, and on the fringe, it was evident that A* search was the best search for the 8-tile puzzle problem. A* search resulted in the least amount of nodes in the memory, and the least amount of nodes explored. Additionally, A* always had the smallest standard deviation throughout the tests. A* outperforms BFS and DLS increase when the number of moves taken to win increases.

References

[1] A. K. Mishra and P. C. Siddalingaswamy, "Analysis of tree based search techniques for solving 8-puzzle problem," in *2017 Innovations in Power and Advanced Computing Technologies (i-PACT)*, Apr. 2017, pp. 1–5. DOI: 10.1109/IPACT.2017.8245012.

[2] S. J. (J. Russell, P. Norvig, and E. Davis, *Artificial intelligence : a modern approach*, Third edition., ser. Prentice Hall series in artificial intelligence. Upper Saddle River, New Jersey: Prentice Hall, 2010, Section: xviii, 1132 pages : illustrations ; 26 cm, ISBN: 0-13-604259-7 978-0-13-604259-4.

5 Appendix

The test cases:

1 0 3 4 2 5 6 7 8
1 2 3 4 0 5 6 7 8
1 2 3 0 4 5 6 7 8
1 2 3 4 0 5 6 7 8
1 2 3 4 5 0 6 7 8
1 2 3 4 0 5 6 7 8
1 2 3 4 7 5 6 0 8
1 2 3 4 0 5 6 7 8
0 1 3 4 2 5 6 7 8
1 2 3 4 0 5 6 7 8
1 3 0 4 2 5 6 7 8
1 2 3 4 0 5 6 7 8
1 2 3 4 7 5 0 6 8

1 2 3 4 0 5 6 7 8
1 2 3 4 7 5 6 8 0
1 2 3 4 0 5 6 7 8
1 3 5 4 2 8 6 7 0
1 2 3 4 0 5 6 7 8
2 3 0 1 4 5 6 7 8
1 2 3 4 0 5 6 7 8
1 2 3 7 0 5 4 6 8
1 2 3 4 0 5 6 7 8
1 2 3 4 5 8 0 6 7
1 2 3 4 0 5 6 7 8
0 1 3 5 2 8 4 6 7
1 2 3 4 0 5 6 7 8
2 3 5 1 0 8 6 4 7
1 2 3 4 0 5 6 7 8
1 3 5 7 2 8 4 6 0
1 2 3 4 0 5 6 7 8
0 3 5 1 4 8 6 2 7
1 2 3 4 0 5 6 7 8
6 1 3 7 2 4 0 8 5
1 2 3 4 0 5 6 7 8
4 1 2 6 7 3 8 5 0
1 2 3 4 0 5 6 7 8
4 1 2 5 0 8 6 3 7
1 2 3 4 0 5 6 7 8
4 3 0 6 1 2 7 8 5
1 2 3 4 0 5 6 7 8