Experimental report for the 2021 COM1005 Assignment: The 8-puzzle Problem*

Thomas Pearson

April 22, 2022

I declare that the answers provided in this submission are entirely my own work. I have not discussed the contents of this assessment with anyone else (except for Heidi Christensen or COM1005 GTAs for the purpose of clarification), either in person or via electronic communication. I will not share the assignment sheet with anyone. I will not discuss the tasks with anyone until after the final module results are released.

1 Descriptions of my breadth-first and A* implementations

1.1 Breadth-first implementation

In order to implement the Breath first search, three new classes were needed.

The first class that was needed was "RunEpuzzleBFS.java", displayed in Figure 1. This class facilitated calling the search operations and linking together the other classes. It implemented such classes as "EpuzzleSearch" and "EpuzzleState" which both played a huge part in the actual running of the search algorithm.

1.1.1 EpuzzleSearch

One of the previously mentioned classes, "EpuzzleSearch", was the first class to be declared inside "RunEpuzzleBFS". Its extends search and its main

*https://github.com/TomPearson38/The8PuzzleAndAStar

goal was to define the goal predicate. In this implementation the goal predicate would be a solved puzzle where each number is in the correct position (The correct positions being, {{1,2,3}, {4,5,6}, {7,8,0}}). The target goal is the only parameter taken into the constructor and is the only value that could be returned from the class when "getTarget()" is ran.

```
8   import java.util.*;
9
10  public class RunEpuzzleBFS{
11
12      public static void main(String[] arg){
13          int[][] startP1 = {{1, 0, 3}, {4, 2, 6}, {7, 5, 8}};
14          int[][] startP2 = {{4, 1, 3}, {7, 2, 5}, {0, 8, 6}};
15          int[][] startP3 = {{2, 3, 6}, {1, 5, 8}, {4, 7, 0}};
16
17          int[][] target = {{1, 2, 3}, {4, 5, 6}, {7, 8, 0}};
18
19          //BFS search algorithm P1
20          EpuzzleSearch searcherP1 = new EpuzzleSearch(target);
21          SearchState initStateP1 = (SearchState) new EpuzzleState(startP1);
22
23          //BFS search algorithm P2
24          EpuzzleSearch searcherP2 = new EpuzzleSearch(target);
25          SearchState initStateP2 = (SearchState) new EpuzzleState(startP2);
26
27          //BFS search algorithm P3
28          EpuzzleSearch searcherP3 = new EpuzzleSearch(target);
29          SearchState initStateP3 = (SearchState) new EpuzzleState(startP3);
30
31          //Search algorithm is ran for each P
32          String resultP1 = searcherP1.runSearch(initStateP1, "breadthFirst");
33          System.out.println(resultP1);
34          String resultP2 = searcherP2.runSearch(initStateP2, "breadthFirst");
35          System.out.println(resultP2);
36          String resultP3 = searcherP3.runSearch(initStateP3, "breadthFirst");
37          System.out.println(resultP3);
38      }
39  }
```

Figure 1: RunEPuzzleBFS Code

1.1.2 EpuzzleState

"EpuzzleState" is the next class to be declared in "RunEpuzzleBFS", being the largest class out of the three. The class extends SearchState and it defines each state in the puzzle. It is what allows us to move from node to node in order to attain the path to the goal predicate. The constructor for the class takes in the current board layout in the form of a 2D integer array. This private variable can be retrieved using the getBoard() public function.

The goalPredicate(), Figure 2, is a public function and checks to see if the current node is the goal node. This is done simply by comparing each

position to see if it matches with the position in the target board. If any position doesn't match then false is returned.

```
/**
* Checks to see if the goal has been achived
* @param searcher Current Searcher
* @return if the presented board matches the goal board
*/
public boolean goalPredicate(Search searcher){
    EpuzzleSearch Esearcher = (EpuzzleSearch) searcher;
    int[][] target = Esearcher.getTarget();

    //loops through loop and compares each element, if any don't match, match variable is set to false
    boolean match = true;
    for(int i = 0; i < target.length; i++){
        for(int j =0; j < target[i].length; j++){
            if(board[i][j] != target[i][j])
                match = false;
        }
    }
    return match;
}
```

Figure 2: GoalPredicate() Code

In order to retrieve the successors the getSuccessors(), Figure 3 function is called. This function returns an array list of SearchStates that are accessible from the current board layout. In order to do this, it first must locate the blank space as the only numbers that can move are the numbers adjacent to the blank space. Once the space is located the if statement lists every possible move, and therefore the associated EpuzzleState. It then adds them to the array list of EpuzzleStates.

When declaring the new puzzleStates, the new board associated with the state is needed and therefore the private function moveBoardPiece() is called. For parameters it takes the coordinates of the blank space and the coordinates of the value being swapped. It then performs a deep copy of the current board and swaps the places, returning the new board layout.

At the end of the function, the EpuzzleStates are converted to SearchStates and returned.

1.1.3 Running of the algorithm

Both of these classes are key in the functional operation of the breath first search and solving the puzzle. In order to start the search, a string is created to save the output and its value is set to the return value from runSearch with the parameters of the initial search state and 'breadthFirst'. The value of the string is then printed to the console.

```java
/**
* Gets the successors for the current node
* @param searcher Current Search
* @return An array list of search states
*/
public ArrayList<SearchState> getSuccessors(Search searcher){
    EpuzzleSearch Esearcher = (EpuzzleSearch) searcher;
    int row = 0;
    int column = 0;

    //Loops through the array to find the position of the blank space
    for(int i = 0; i < board.length; i++){
        for(int j =0; j < board[i].length; j++){
            if(board[i][j] == 0){
                row = i;
                column = j;
            }
        }
    }

    //Successor arrays are declared
    ArrayList<EpuzzleState> puzList = new ArrayList<EpuzzleState>();
    ArrayList<SearchState> statList = new ArrayList<SearchState>();

    //Depending on the position of the 0, there are moves that are able to be made. This if statement finds the available moves.
    if(row == 0 && column == 0){
        puzList.add(new EpuzzleState(moveBoardPiece(row,column,0,1)));
        puzList.add(new EpuzzleState(moveBoardPiece(row,column,1,0)));
    }
    else if(row == 0 && column == 1){
        puzList.add(new EpuzzleState(moveBoardPiece(row,column,0,0)));
        puzList.add(new EpuzzleState(moveBoardPiece(row,column,0,2)));
        puzList.add(new EpuzzleState(moveBoardPiece(row,column,1,1)));
    }
```

Figure 3: Start of getSuccessors() Code

1.2 A* implementation

For the A* search algorithm a few changes, as well as additions, to the code had to be implemented. Firstly, "EpuzzleState"'s constructor contained two new parameters, as seen in Figure 4. The first was local cost, which informed the search algorithm the cost of moving to that position. Secondly, a Boolean variable called hamming. Hamming purpose was to inform the program in how to calculate the local cost of the node. If hamming was set to true, Hamming search would be used. If it was set to false Manhattan search would be used instead.

Now as the search relies heavily on the new local cost it would need to be calculated for every successor for the current node in order to determine which now would be visited next.

```
9  public class EpuzzleState extends SearchState {
10     private int[][] board;
11     private boolean hamming; //True if hamming search is used, false if manhattan search
12
13     /**
14     * Constructor
15     * @param b Current Board Layout
16     * @param lc Local Cost
17     * @param h Hamming Search
18     */
19     public EpuzzleState(int[][] b, int lc, boolean h){
20         board = b;
21         localCost = lc;
22         hamming = h;
23     }
```

Figure 4: Start of the new EPuzzleState() Code

1.2.1 Hamming Cost Method

The first way the remaining cost would be calculated would be with the Hamming method, figure 5. The method counted the number of places that were out of place compared to the final target and would increment the value by one each time it found one out of place.

```
if(hamming == true){
    //Hamming
    //Each position of the board is compared to see if it contains the correct value
    //If it does then the counter is increased
    int hamCount = 0;
    for(int i = 0; i < target.length; i++){
        for(int j =0; j < target[i].length; j++){
            if(board[i][j] != target[i][j])
                hamCount++;
        }
    }
    //THe value of the counter is returned.
    return hamCount;
}
```

Figure 5: Hamming Cost Model Code

1.2.2 Manhattan Cost Method

The Manhattan method requires a bit more calculation compared to the Hamming method, figure 6. This method focuses around how many places the numbers are out of place by. So if it would take two moves to move a piece to the correct position the Manhattan cost for that individual place would be 2. For the total of the board's local cost, the cost for each position would be added together.

The algorithm for this calculation first requires the correct position to be found for each piece on the current board. This is done by the private function findCorrectPos(). The coordinates of the correct position are then compared to the incorrect position and the magnitude of the difference between the coordinates, for both the row and column, is added together to arrive at a value indicating the number of moves it would take to get the piece in the correct position.

```
//Manhattan
//The sum of the number of spaces each position is out of position by is the value here.
int manCount = 0;
for(int i = 0; i < target.length; i++){
    for(int j =0; j < target[i].length; j++){
        //If the position is out of place
        if(board[i][j] != target[i][j])
        {
            //Correct position is found and returned as an array
            //correctPos[0] = row
            //CorrectPos[1] = column
            int[] correctPos = findCorrectPos(board[i][j], target);

            //The magnitude of each calculation is added to the total value.
            if(i > correctPos[0]){
                manCount += (i - correctPos[0]);
            }
            else{
                manCount += (correctPos[0] - i);
            }

            if(j > correctPos[1]){
                manCount += (j - correctPos[1]);
            }
            else{
                manCount += (correctPos[1] - j);
            }
        }
    }
}
//The magnitude is then returned
return manCount;
```

Figure 6: Manhattan Cost Model Code

2 Results of assessing efficiency for the two search algorithms

The test hypothesis states that: **"A* is more efficient than breath-first, and the efficiency gain is greater the more difficult the problem and the close to the estimates are to the true cost."**

In order to test this hypothesis a number of tests were completed.

2.1 Pre-defined Tests

The first tests ran focused around pre-defined tests that were listed in the assignment brief. Figure 7 shows the starter configurations for the tests and the goal configuration. This figure was taken from the assignment brief.

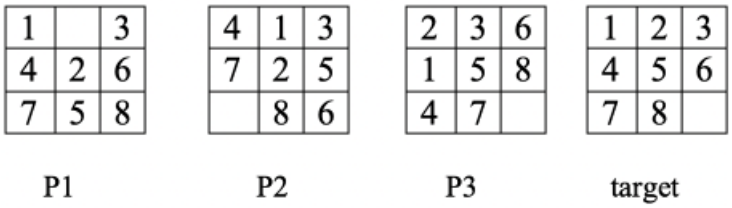

Figure 7: Test Patterns for BFS and the target configuration. Taken from the assignment brief

These test configurations were completed by each algorithm. Table 1 displays the results from the completion of the algorithm. The efficiency is shown as a decimal where:

$$Efficiency = \frac{NodesOnSolutionPath}{NumberOfNodesVisited}$$

Test	BFS	A*(Hamming)	A*(Manhattan)
P1	0.21052632	0.36363637	0.4
P2	0.12280702	0.23333333	0.20588236
P3	0.05625	0.14285715	0.15789473

Table 1: Pre-defined Tests Results Table

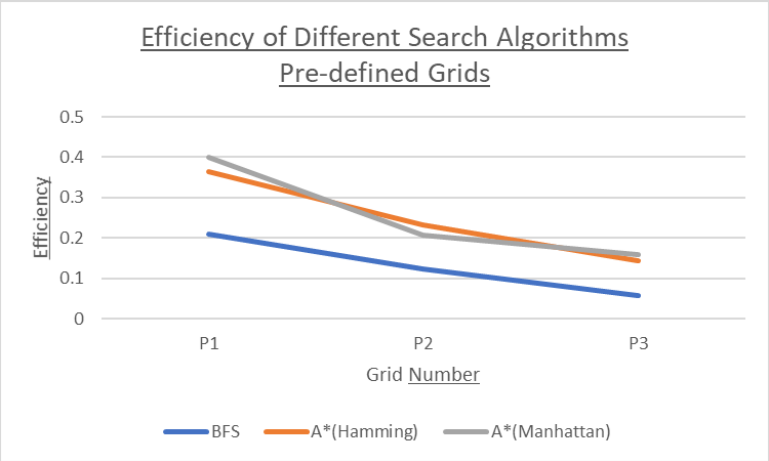

Figure 8: Pre-defined Tests Results Graphical Representation

From the starter test is becomes apparent that the A* search is already quicker than BFS. However in order to prove that these results were not anonymous more tests would need to be created and ran.

2.2 Generated Tests

In order to create more tests the provided function EpuzGen was implemented. This meant that the RunSearch file for the searches would need to be edited to accommodate for seed input. Random seeds were then supplied to each with a pre-defined difficulty for the puzzle generation. Both the seeds and the difficulty were then varied in order to generate the results. Tables 2 and 3 show the results obtained from different difficulties of generation when the seed supplied changed. Figures 9 and 10 display these results in a graphical format.

Seed - Difficulty 6	BFS	A*(Hamming)	A*(Manhattan)
12345	0.002893309	0.008791209	0.0093622
23456	0.000717306	0.001700224	0.003529412
45678	0.000972962	0.001205966	0.003000158
56789	0.005595523	0.026871402	0.035714287
14725	0.000542341	0.002783156	0.004138179
24680	0.002769124	0.020565553	0.021333333
13579	0.003015454	0.00679983	0.009417304
12389	0.005103901	0.015904572	0.024844721
38388	0.005709625	0.019830028	0.03125
64219	0.001714747	0.009322866	0.009396637
99999	0.001967593	0.008717949	0.01559633

Table 2: Difficulty 6 Seed Results

Seed - Difficulty 8	BFS	A*(Hamming)	A*(Manhattan)
12345	0.000462973	0.005954069	0.008713693
82497	0.000441316	0.001470691	0.002423543
67891	0.001492144	0.005232379	0.006374203
38388	0.001873691	0.003504432	0.01010101
54545	0.001562356	0.008106819	0.010493827
13579	0.003015454	0.00679983	0.009417304
38388	0.005709625	0.019830028	0.03125
64219	0.001714747	0.009322866	0.009396637
99999	0.001967593	0.008717949	0.01559633

Table 3: Difficulty 8 Seed Results

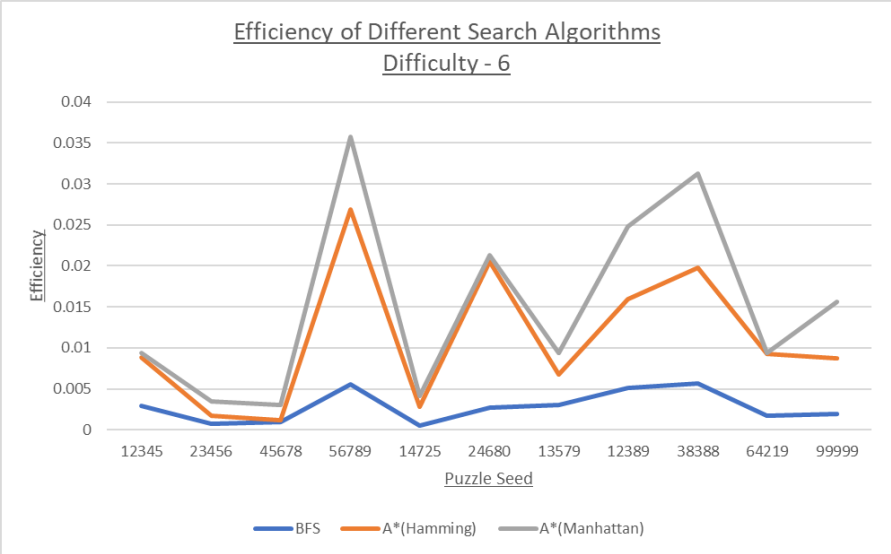

Figure 9: Results for Difficulty 6 Calculations

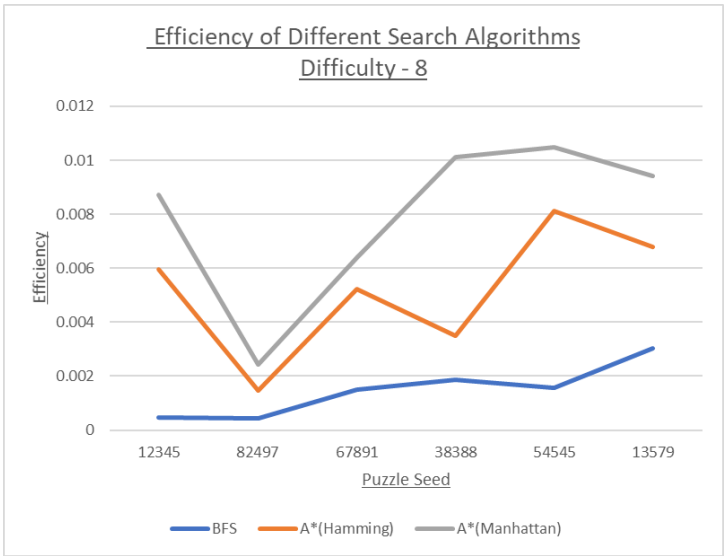

Figure 10: Results for Difficulty 8 Calculations

3 Conclusions

Running a variety of tests with ranging difficulty it has become easily apparent that the Breadth First Search strategy is inefficient when compared to the A* search algorithm. We can see this very clearly on the graphs where in every case at least one A* search method was faster than breadth first.

Furthermore, the Manhattan method, beat if not equaled the efficiency of the Hamming method of calculating remaining cost in every calculation. This is because of the larger range of values of local cost, that can be produced and compared with each other. The Hamming method is limited to a range of 9 numbers and the Manhattan method far surpasses this limitation, allowing it to clearly demonstrate which moves have a lower local cost.