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

Usman Hussain

May 4, 2022

Declaration: 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

For my breadth-first implementation, I created 3 new classes, EPuzzleState, EPuzzleSearch and RunEPuzzleSearch (which is a test class). In EPuzzleState, I made methods to determine:

- If the current state was equal to the target (if this was the case, we conclude the search)
- If a successor to the current state was previously found as a successor (the search engine would handle the results of this)
- Successors of the current state

My main job was to implement the getSuccessors() method for 8 puzzle. To determine the successors of a state, there are only a maximum of 4 possible successors to a certain state (after right, left, up or down moves). I created

*My *PRIVATE* Github URL: https://github.com/usman1239/com1005MachinesAI

separate methods for each of these moves which would return the state, if this move was possible. A move was possible and would return a non-null puzzle state if it met a condition e.g. to move an element into the space using a left move, I would need to check if there was an element to the right of the empty space to slide in. For example, Figure 1 is my Java code for moving an element from the left into the empty space. After this, I returned the successors as a list for the search engine to handle.

```
public EPuzzleState rightMove(int[][] s, int[] emptyCoordinates) {
    //first checks whether a number can be moved to the right into the empty space
    //if not, returns null
    if (emptyCoordinates[0] == 0) {
        return null;
    }
    else {
        //returns a copy of the current array puzzle
        int[][] toReturn = duplicatedArray(s);
        //'moves' the number right, into the empty slot
        int zero = toReturn[emptyCoordinates[1]] [emptyCoordinates[0]];
        toReturn[emptyCoordinates[1]] [emptyCoordinates[0]] = toReturn[emptyCoordinates[1]] [emptyCoordinates[0] - 1];
        toReturn[emptyCoordinates[1]] [emptyCoordinates[0] - 1] = zero;
        return new EPuzzleState(toReturn);
    }
}
```

Figure 1: Code showing an element moving in the empty space from the left

1.2 A* implementation

For A* implementation, I used a similar approach to my BFS one. There was 1 extra class we were provided with (EPuzzGen) which was used to generate puzzles that were possible, so this was used to test my implementation and gather results to argue the hypothesis. Due to A* requiring a local cost (which was always 1) and a estimated remaining costs on top of the array with the puzzle, the EPuzzleState constructor now included code for this. There were not many changes to the successors apart from the costs being calculated and a key part to each state.

```
public EPuzzleStateAStar(int[][] c, int lc, int erc){
    currentState = c;
    localCost = lc;
    estRemCost = erc;
}
```

Figure 2: EPuzzleState constructor for A*

The two methods to calculate the estimated remaining cost (estRemCost) were:

- Hamming
- Manhattan

(In the code for A* implementation, I have commented out 1 of the estRemCost line calculations in each of the move methods (e.g. in rightMove). You can comment accordingly for the approach you require).

A Hamming distance is a sum for how many elements of the 8 puzzle are out of place when compared to the target puzzle. In my implementation for this, I created a separate method which looped through a puzzle array and kept a count for how many numbers were out of place when compared to the target. I did not include the empty space when working out the Hamming value, so a possible maximum of 8 for 1 state. This was relatively simple to create and my code for this is below:

```
private int hamming(int[][] src, int[][] t) {
    int numOutOfPlace = 0;
    for (int x = 0; x < src.length; x++) {
        for (int y = 0; y < src[x].length; y++) {
            if (src[x][y] != t[x][y] && src[x][y]!=0) {
                numOutOfPlace++;
            }
        }
    }
    return numOutOfPlace;
}
```

Figure 3: Estimated remaining cost: Hamming Approach

A Manhattan distance is a sum of distances of each element from its correct position in the 8 puzzle. In my implementation for this, I created another method. My approach was to find the index of the element in the current state and also in the target, then use an absolute distance to calculate the distance that element has to move within the puzzle. To do this, I used for loops and a total counter, which was a sum of all the distances found. Compared to the implementation for Hamming distance, this was a lot harder to do. (As for Hamming, I did not include the empty space on this distance).

```
//exclude the empty space
for (int x = 1; x < 9 ; x++) {
    int i;
    int j;
    for (i = 0; i < 3; i++) {
        for (j = 0; j < 3; j++) {
            if (s[i][j] == x) {
                si = i;
                sj = j;
            }
        }
    }
    for (i = 0; i < 3; i++) {
        for (j = 0; j < 3;j++) {
            if (t[i][j] == x) {
                ti = Math.abs(i-si);
                tj = Math.abs(j-sj);
                totalManhattan += ti + tj;
            }
        }
    }
}
return totalManhattan;
```

Figure 4: Estimated remaining cost: Manhattan Approach (logic only)

2 Results of assessing efficiency for the two search algorithms

Hypothesis: *A* is more efficient than breadth-first, and the efficiency gain is greater the more difficult the problem and the closer the estimates are to the true cost.*

To obtain my results, I used a seed of 9908 across 4 different difficulties. As per my results, it is valid to assume that A* searching is more efficient than breadth-first. For each difficulty I tested, the average efficiency for A* was much higher than it was for the average efficiency of BFS.

As for the efficiency gain being greater the more difficult the problem is, this was mostly true from my results. Looking at each of the tables, we can see that the efficiency gain for A* is a lot better because as we increase the difficulty, the A* searching average efficiency does not drop by as much as BFS, which suggests that as the difficulty increases, BFS takes longer to find the best solution. This means that in terms of efficiency gain, A* is gaining quite a lot in terms of efficiency the more difficult the puzzle, a contrast to BFS. For the difference in average efficiencies in A* for difficulties 10 and 12 (Tables 3 and 4), we can see that there is a massive drop, which suggests that the gain between these 2 difficulties is not as high as from BFS. However, there is 1 interesting point to note because from Table 3, the average efficiency for both A* types is the highest for this difficulty out of all

results collected. This is likely due to the result which provided an efficiency of 1, and could prove to be an anomaly. If we take this into account, then it is fair to assume that the efficiency gain may be continuously better for A* than for BFS.

Finally, regarding the closeness of estimates to the true cost, we can compare the difference between both A* methods used, Hamming and Manhattan. Manhattan and Hamming are both ways to find the distance from the current state to the target state, however Manhattan will give a more accurate value than Hamming. Manhattan uses absolute values and in my code, I have used this as well. This is very important and will provide a much closer estimate than Hamming, which simply counts how many elements are out of place. This doesn't provide a complete representation of what the best next move would be.

For each of the tables, I had calculated a percentage difference between the average Hamming efficiency and average Manhattan efficiency. For difficulties 6 and 8, we can see the difference is increasing and also for difficulty 12. Therefore, it is fair to assume that Manhattan provides a better solving efficiency than Hamming and that the closer the estimates to the true cost, the better the efficiency gain. The percentage difficulty for Table 3 is likely due to the puzzle where both efficiencies were 1.

Puzzle	BFS	Hamming	Manhattan
163458720	0.00443787	0.053956833	0.13043478
132540786	0.001340882	0.015138772	0.04118993
123654870	4.5072116E-4	0.005839822	0.026819924
213450876	1.8201346E-4	0.0033142513	0.012366498
412653780	0.041860465	0.2682927	0.57894737
143526780	0.0022172949	0.024425287	0.05964912
513420786	0.009623095	0.14117648	0.22222222
423586170	0.0023322816	0.028764805	0.08585858
523186470	0.011754069	0.12745099	0.26
163420785	0.002044206	0.03168317	0.07511737
Average	0.007624289812	0.07000431103	0.1492605792

Table 1: 7 random seeds tested with difficulty 6, and efficiencies for the respective search type

Percentage Difference between Hamming and A* = 72.297%

Puzzle	BFS	Hamming	Manhattan
413865720	0.0019148457	0.026397515	0.0867347
163458720	0.0092370112	0.053956833	0.13043478
132540786	0.041860465	0.015138772	0.04118993
153428076	0.0081211965	0.025	0.062730625
264153780	0.008469055	0.14444445	0.35135135
532146870	0.003129056	0.012820513	0.05322129
316420785	7.9184126E-4	0.017094018	0.06185567
132475860	0.0022383146	0.025993884	0.07359307
412356870	0.0026666668	0.013552069	0.055393588
123654870	4.5072116E-4	0.005839822	0.026819924
Average	0.007887917322	0.0340237876	0.0943324927

Table 2: 7 random seeds tested with difficulty 8, and efficiencies for the respective search type

Percentage Difference between Hamming and A* = 93.9708 %

Puzzle	BFS	Hamming	Manhattan
382176450	8.186832E-4	0.013718411	0.06690141
103452768	0.001059322	0.071794875	0.1590909
120435687	4.1066157E-4	0.005970998	0.03125
456283170	8.026699E-4	0.0028262471	0.017424243
162053784	0.0012266594	0.016559338	0.05882353
152740386	0.0010109519	0.081871346	0.3783784
413865720	0.0054815975	0.026397515	0.0867347
482516703	1.9345054E-4	0.0036393714	0.021012416
013526478	0.009754358	1.0	1.0
125360487	0.0026459802	0.018126888	0.074074075
Average	0.002340433421	0.12409049895	0.1893689674

Table 3: 7 random seeds tested with difficulty 10, and efficiencies for the respective search type

Percentage Difference between Hamming and A* = 41.65 %

Puzzle	BFS	Hamming	Manhattan
320564178	9.422266E-4	0.015019763	0.09047619
382176450	8.186832E-4	0.013718411	0.06690141
341026758	0.002738788	0.043243244	0.1632653
152340687	1.9678583E-4	0.0017789637	0.01563518
103824756	0.0054815975	0.071794875	0.1590909
273506418	0.0029591636	0.05033557	0.15789473
142856307	2.9911217E-4	0.003717472	0.021956088
273410568	4.3246115E-4	0.0079302145	0.04597701
861423570	5.0226017E-4	0.006818182	0.066246055
184756230	1.8180496E-4	0.0013922148	0.011116051
Average	0.001455288318	0.021574891	0.0798558914

Table 4: 7 random seeds tested with difficulty 12, and efficiencies for the respective search type

Percentage Difference between Hamming and A* = 114.918%

3 Conclusions

Conclusions I have drawn from my experimental work are:

- A* is much more efficient than BFS in most cases, especially as the difficulty of the problem increases.
- The efficiency gain for A* as difficulty of a problem increases is higher than the efficiency gain for BFS.
- Manhattan distance provides a closer estimate to the true cost than Hamming distance does, therefore proving more efficient in most cases.

In the event that I did not get a potential anomaly, I believe that my results would have fully agreed with the hypothesis. Nevertheless, they provide a good representation of what was expected from this experiment, which was mostly agreeing with the hypothesis.