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

Jai Mistry

April 18, 2022

1 Descriptions of my breadth-first and A* implementations

1.1 Breadth-first implementation

The Breadth-first implementation(BFS) is a search where every single possible route is explored until the target is reached. It does this by checking all the child nodes of the starting point and it keeps checking all of the generations afterwards until it reaches the target. In the case of the 8 puzzle search, BFS would look at all the possible values that can move into the empty space.

```
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        if (seed[i][j] == 0) {
            // Creates copies of the current state for movement
            int[][] rightSeed = copyState(seed);
            int[][] leftSeed = copyState(seed);
            int[][] upSeed = copyState(seed);
            int[][] downSeed = copyState(seed);

            if (j + 1 < 3) {
                // Moves empty to the right
                swap(i, j, i, j + 1, rightSeed);
                eslis.add(new EpuzzleState(rightSeed));
            }

            if (j - 1 >= 0) {
                // Moves empty to the left
                swap(i, j, i, j - 1, leftSeed);
                eslis.add(new EpuzzleState(leftSeed));
            }

            if (i - 1 >= 0) {
                // Moves empty up
                swap(i, j, i - 1, j, upSeed);
                eslis.add(new EpuzzleState(upSeed));
            }

            if (i + 1 < 3) {
                // Moves empty down
                swap(i, j, i + 1, j, downSeed);
                eslis.add(new EpuzzleState(downSeed));
            }
        }
    }
}
```

Figure 1: This is the code which looks at the child nodes.

https://github.com/Jplayz/8Puzzle

Figure 1 shows how the code does this. The code checks the values in the 2D matrix to see where the value of the empty tile (zero) is, and depending on the position of the empty it will decide if there is a valid move to be made in the upward, left, right or downwards direction. It then adds all of the possible values to an arraylist which will then run through again but using one of those as a starting position until it finds the target.

1.2 A* implementation

For A* implementation, I had to add costs to the search. To keep it simple every move had a cost of one, as one tile would be moved at a time but A* needs an estimated remaining cost and that cost was calculated two different ways, Hamming and Manhattan.

Hamming

Hamming is calculated by looking at all of the out of place tiles to calculate an estimated remaining cost. This was simple to implement as if the value of say [0,0] was three for the initial seed it would just be checked against [0,0] of the target, which normally would be one, and then see that it was false so it would move on.

```
public int hamming(Search searcher) {
    EpuzzleSearch esearcher = (EpuzzleSearch) searcher;
    int[][] tar = esearcher.getTarget(); // get target
    int hamming = 9;

    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            if (tar[i][j] == seed[i][j])
                hamming -= 1;
        }
    }
    return hamming;
}
```

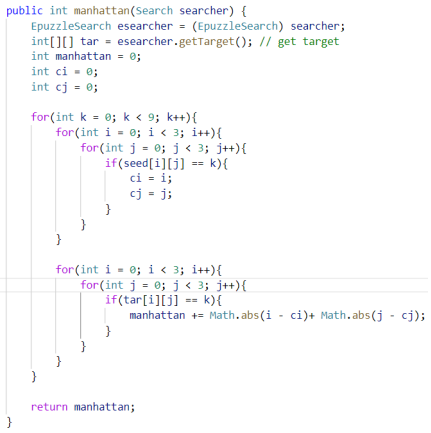
Figure 2: This is the code to calculate the Hamming value.

The code implemented in Figure 2 assumes that every value is out of place so it starts with a value of nine. If a correct value is found it will reduce the counter and then return the estimated remaining cost.

Manhattan

Manhattan is calculated by seeing how far each tile is out of place by. This was more difficult to implement as it required some calculations to figure out how far away the seed is from the target. In this section I want to describe what I added to my classes to implement the A* search engine

for the 8-puzzle problem. This should include a description of how I have implemented the two distance measures. Screenshots could also be great in this section.



```
public int manhattan(Search searcher) {
    EpuzzleSearch esearcher = (EpuzzleSearch) searcher;
    int[][] tar = esearcher.getTarget(); // get target
    int manhattan = 0;
    int ci = 0;
    int cj = 0;

    for(int k = 0; k < 9; k++){
        for(int i = 0; i < 3; i++){
            for(int j = 0; j < 3; j++){
                if(seed[i][j] == k){
                    ci = i;
                    cj = j;
                }
            }
        }

        for(int i = 0; i < 3; i++){
            for(int j = 0; j < 3; j++){
                if(tar[i][j] == k){
                    manhattan += Math.abs(i - ci)+ Math.abs(j - cj);
                }
            }
        }
    }

    return manhattan;
}
```

Figure 3: This is the code to calculate the Manhattan value.

The code in Figure 3 runs many loops but the main loop goes from zero to eight which represents all the tiles. The first sub-loop, finds the current position of value k and stores it, as ci and cj. The second loop looks at the target to find that value k and then it compares how far the positions are using Equation 1.

$$Manhattan = Manhattan + |i - ci| + |j - cj| \tag{1}$$

2 Results of assessing efficiency for the two search algorithms

Here I will describe the results of the experiments I had to run to test my hypothsis.

Table 1 shows something super important. This is an example of a table in LaTeX and how you might reference it in the text.

3 Conclusions

Here I need to write what conclusions I can draw from my experimental work.

System	Measurement [%]
Basic	40%
Improved v1	42%
Improved v2	43%

Table 1: Example table