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

Noor Elreidy

May 3, 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.

Contents

[*]https://github.com/nelreidy/com1005.git

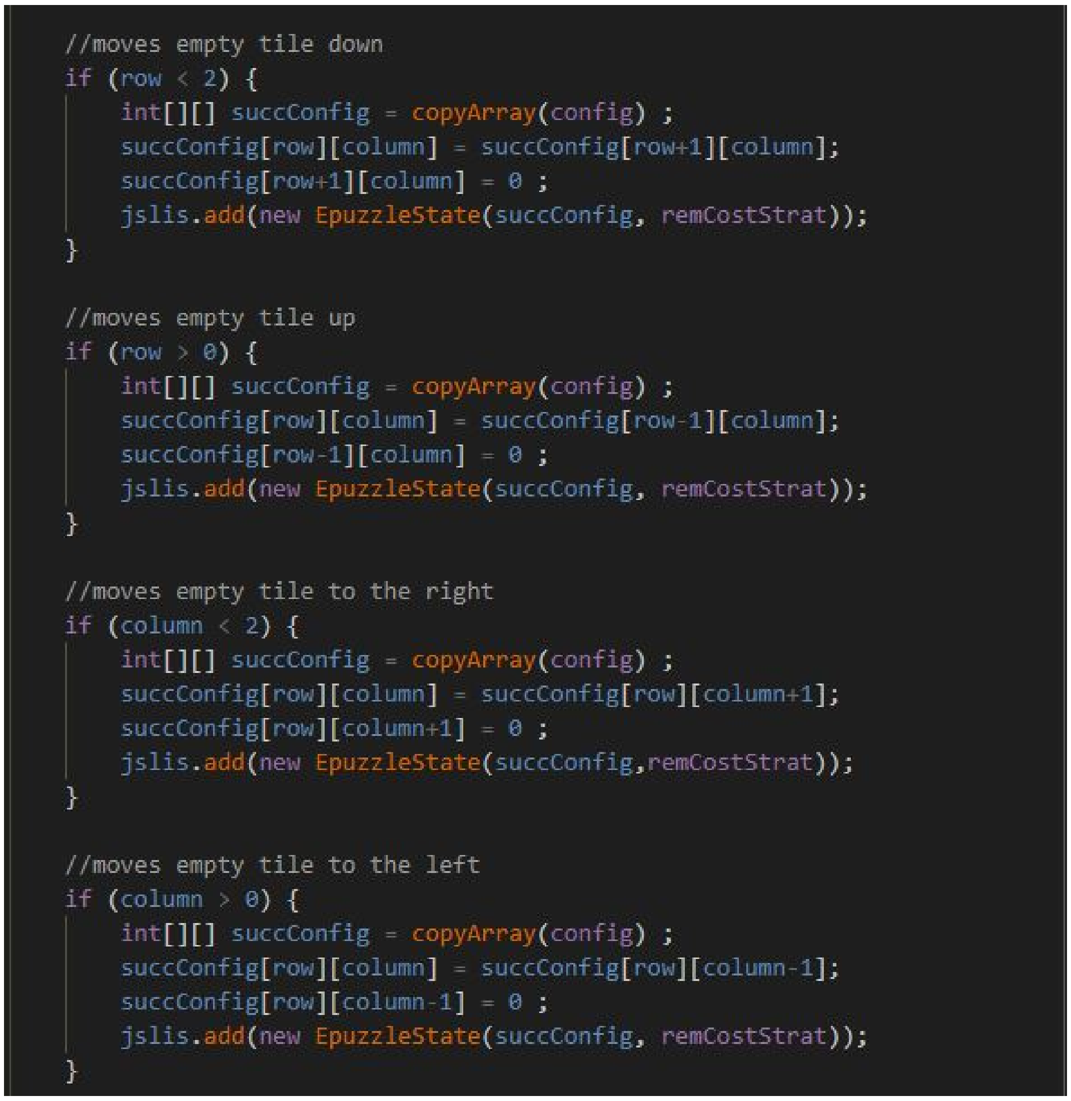
1 Descriptions of my breadth-first and A* implementations

1.1 Breadth-first implementation

The implementation of the breadth-first search required the creation of two additional classes, **EpuzzleSearch** and **EpuzzleState**, which extend the classes Search and SearchState, respectively. There is also the class **RunEpuzzleBFS**. It contains a main method used for running the breadth-first search on the 8-puzzles *P1*, *P2* and *P3* which are specified in the assignment brief.

Starting with **EpuzzleSearch**, it is a short class that allows the initialisation of a Search object and sets the target configuration of the 8-puzzle. Since there is only one target configuration for 8-puzzles, it is hard-coded. The needed functionality for a search is carried out by its super class **Search**. By invoking the method runSearch on an EpuzzleSearch object, and passing an initial state of type SearchState and the string "breadthFirst" as parameters, a breadth first search is run on the initial state 8-puzzle.

EpuzzleState stores an 8-puzzle configuration as a 2D array. Its method getSuccessors returns a list of the successor states of the current state. Figure 1 shows the four conditions used to obtain new successor states after the postion of the empty tile is determined and stored in variables *row* and *column*. This class also has the method *sameState* which is used to find whether a successor state has already been encountered in **Search.java**, as well as *goalPredicate* which returns true if the current state is the target state.

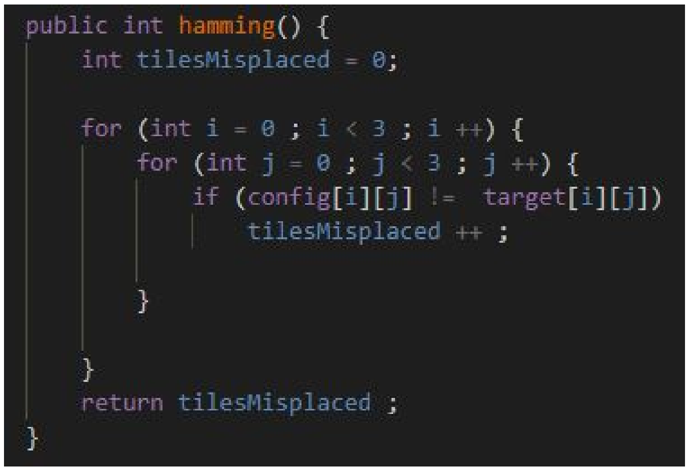


```java
//moves empty tile down
if (row < 2) {
    int[][] succConfig = copyArray(config) ;
    succConfig[row][column] = succConfig[row+1][column];
    succConfig[row+1][column] = 0 ;
    jslis.add(new EpuzzleState(succConfig, remCostStrat));
}

//moves empty tile up
if (row > 0) {
    int[][] succConfig = copyArray(config) ;
    succConfig[row][column] = succConfig[row-1][column];
    succConfig[row-1][column] = 0 ;
    jslis.add(new EpuzzleState(succConfig, remCostStrat));
}

//moves empty tile to the right
if (column < 2) {
    int[][] succConfig = copyArray(config) ;
    succConfig[row][column] = succConfig[row][column+1];
    succConfig[row][column+1] = 0 ;
    jslis.add(new EpuzzleState(succConfig,remCostStrat));
}

//moves empty tile to the left
if (column > 0) {
    int[][] succConfig = copyArray(config) ;
    succConfig[row][column] = succConfig[row][column-1];
    succConfig[row][column-1] = 0 ;
    jslis.add(new EpuzzleState(succConfig, remCostStrat));
}
```

Figure 1: Part of getSuccessors() in EpuzzleState.java

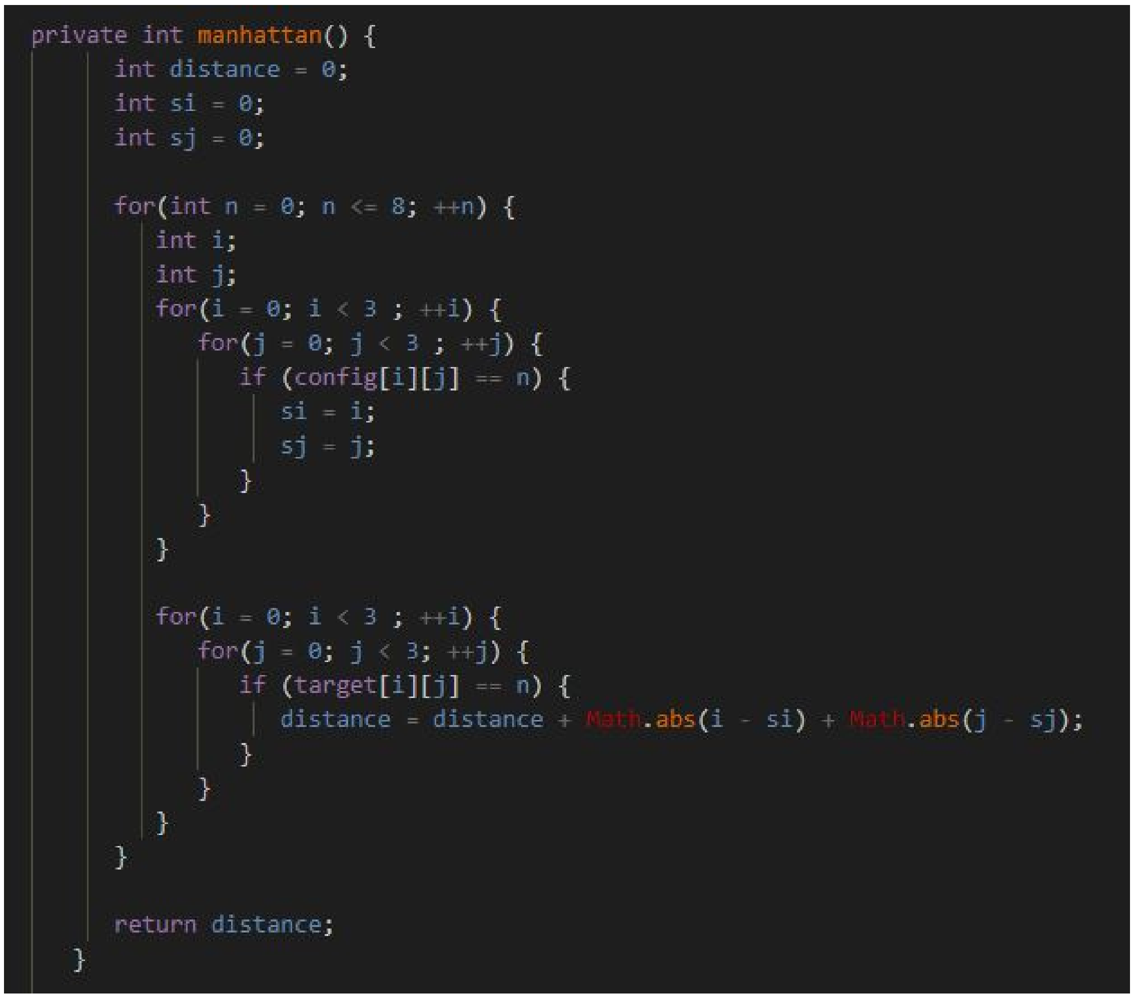
1.2 A* implementation

The implementation of the A* search also required the same classes as the breadth-first search. The **EpuzzleSearch** class is exactly the same as the one for breadth-first. The **EpuzzleState** class is different, however. It takes an additional parameter in constructor,the strings *"hamming"* or *"manhattan"* which specify the strategy used for calculating the estimated remaining cost. Figures 2 and 3 show the methods' implementation. The estimated remaining cost is stored in the variable *estRemCost*, while the local cost is hard-coded as 1. Both the local and *estRemCost* are used in **Search.java** to establish the appropriate Search route.



```
public int hamming() {
    int tilesMisplaced = 0;

    for (int i = 0 ; i < 3 ; i ++) {
        for (int j = 0 ; j < 3 ; j ++) {
            if (config[i][j] !=  target[i][j])
                tilesMisplaced ++ ;

        }

    }
    return tilesMisplaced ;
}
```

Figure 2: hamming() in SearchEngine/Astar/EpuzzleState.java



```java
private int manhattan() {
    int distance = 0;
    int si = 0;
    int sj = 0;

    for(int n = 0; n <= 8; ++n) {
        int i;
        int j;
        for(i = 0; i < 3 ; ++i) {
            for(j = 0; j < 3 ; ++j) {
                if (config[i][j] == n) {
                    si = i;
                    sj = j;
                }
            }
        }

        for(i = 0; i < 3 ; ++i) {
            for(j = 0; j < 3; ++j) {
                if (target[i][j] == n) {
                    distance = distance + Math.abs(i - si) + Math.abs(j - sj);
                }
            }
        }
    }

    return distance;
}
```

Figure 3: manhattan() in SearchEngine/Astar/EpuzzleState.java

2 Results of assessing efficiency for the two search algorithms

2.1 P1, P2 and P3

For starters, the breadth-first and A* search strategies were run on the 8-puzzles *P1*, *P2* and *P3* (which are in ascending order of difficulty). Both the hamming and manhattan strategies for calculating estimated remaining cost for A* were compared as well. Table 1 shows the results. Breadth-first performs substantially worse as difficulty increases, with very low efficiency. A* performs much better with hamming surpassing manhattan by a small portion.

Puzzle	Strategy	Efficiency	Iterations
P1	Breadth-first	0.400	10
	A*(Hamming)	1.000	4
	A*(Manhattan)	1.000	4
P2	Breadth-first	0.130	54
	A*(Hamming)	1.000	7
	A*(Manhattan)	0.875	8
P3	Breadth-first	0.055	165
	A*(Hamming)	0.900	10
	A*(Manhattan)	0.750	12

Table 1: P1,P2 and P3 results

2.2 Randomly generated puzzles with varying difficulties

This section tests various seeds with difficulties 4,6,8,10 and 12 to test the hypothesis *A* is more efficient than breath-first, and the efficiency gain is greater the more difficult the problem and the closer the estimates are to the true cost.* The class **EpuzzGen** is used to generate random puzzles at varying difficulties, and the seeds are used to create the same puzzle for the 3 strategies.

2.2.1 Seed 123

Table 2 shows the results of running the 3 different search strategies for the seed 123. Although there is an inconsistency in the results where difficulty 12 takes fewer iterations and is more efficient than the previous 2 difficulties, it is consistent with the hypothesis. For all difficulties, A* Manhattan is the most efficient with the lowest number of iterations while breadth-first is the least efficient with the highest number of iterations. For difficulty 6, A* Manhattan is 24 times as efficient as breadth-first while Hamming is 13 times as efficient. For difficulty 12, Manhattan is 46 times as efficient as breadth-first , while Hamming is 15 times as efficient.

Difficulty	Strategy	Efficiency	Iterations
	Breadth-first	0.0047	2965
6	A*(Hamming)	0.0642	218
	A*(Manhattan)	0.1148	122
	Breadth-first	0.0007	25649
8	A*(Hamming)	0.0124	1537
	A*(Manhattan)	0.0318	597
	Breadth-first	0.0007	29359
10	A*(Hamming)	0.0069	2880
	A*(Manhattan)	0.0260	768
	Breadth-first	0.0018	9223
12	A*(Hamming)	0.0279	609
	A*(Manhattan)	0.0837	203

Table 2: Difficulties 6,8,10,12 for seed 123

2.2.2 Seed 3563

Table 3 shows the results of running the 3 different search strategies for the seed 3563 . This time the results are consistent for all difficulties where efficiency decreases and number of iterations increases as difficulty increases. Here the results also support our hypothesis. For example, difficulty for A Star Manhattan difficulty 12 is 73 times as efficient as breadth-first while Hamming is 11 times as efficient.

Difficulty	Strategy	Efficiency	Iterations
	Breadth-first	0.5000	6
6	A*(Hamming)	1.0	3
	A*(Manhattan)	1.0	3
	Breadth-first	0.0019	8225
8	A*(Hamming)	0.0360	445
	A*(Manhattan)	0.0860	186
	Breadth-first	0.0005	41648
10	A*(Hamming)	0.0081	2472
	A*(Manhattan)	0.0302	662
	Breadth-first	0.0002	96292
12	A*(Hamming)	0.0022	10689
	A*(Manhattan)	0.0147	1562

Table 3: Difficulties 6,8,10,12 for seed 3563

2.2.3 Seed 13624

Table 4 shows the results of running the 3 different search strategies for the seed 13624 . This seed's puzzles are the most difficult among the ones I have tested . Here we can clearly see the difference between Breadth-first and A star, especially Manhattan. For example, difficulty 12's Manhattan is 51 times as efficient as breadth-first .

Difficulty	Strategy	Efficiency	Iterations
6	Breadth-first	0.0043	3494
	A*(Hamming)	0.0507	296
	A*(Manhattan)	0.1087	138
8	Breadth-first	0.0008	25265
	A*(Hamming)	0.0135	1404
	A*(Manhattan)	0.0438	434
10	Breadth-first	0.0007	28785
	A*(Hamming)	0.0075	2655
	A*(Manhattan)	0.0226	886
12	Breadth-first	0.0002	139551
	A*(Hamming)	0.0014	18193
	A*(Manhattan)	0.0103	2433

Table 4: Difficulties 6,8,10,12 for seed 13624

3 Conclusions

These experiments shows that, all in all, the A* search strategy is much more efficient and admissible than breadth-first. The remaining cost estimation strategy *Hamming* seems to do slightly better at the beginning when easy puzzles are provided, but as difficulty increases, the *Manhattan* strategy surpasses *Hamming* by an extensive proportion. Breadth-first faces a problem of combinatorial explosion which makes it perform poorly as difficulty increases. On the other hand, Hamming is not an accurate enough estimate of remaining cost, which also makes it perform poorly compared to Manhattan. These findings confirm our aforementioned hypothesis.