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 i

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

YI LI

May 4, 2022

1 Descriptions of my breadth-first and A* implementations

1.1 Breadth-first implementation

The BFS implementation depends on EpuzzleState, which has goalPredicate, sameState and getSuccessors. The first two determine if the selected nodes are the goal or same. The Epuzzle numbers(0 8) are stored in the array.

```
public boolean sameState(SearchState s) {
    EpuzzleState epState = (EpuzzleState) s;
    for(int i=0; i<9; i++) {
        if(puzzBoard[i] != epState.getPuzzBoard()[i]) {
            return false;
        }
    }
    return true;
}
```

Figure 1: sameState method in BFS

```
public boolean goalPredicate(Search searcher) {
    EpuzzleSearch epSearcher = (EpuzzleSearch) searcher;
    for(int i=0; i<epSearcher.getSize(); i++) {
        if(epSearcher.getGoal()[i]!=puzzBoard[i]) {
            return false;
        }
    }
    return true;
}
```

Figure 2: goalPredicate method in BFS

*private github url:https://github.com/momoyi0929/Com1005$_{A}ssignment_{2}022_{Y}ILI.git$

The most significant method is getSuccessors. It works with getSpace and changePosition method.

```
public int getSpace(int[] board) {
    for(int i=0; i<9; i++) {
        if(board[i] == 0) {
            space = i;
        }
    }
    return space;
}
```

Figure 3: setSpace method in BFS

Figure 3: When invoking getSpace, it will assign the index of 0 to the space variable and we will know where the 0 number exactly is, when we invoke space.

```
public void changePosition(int n1, int n2, ArrayList<EpuzzleState> eps) {
    //copy a new one Epuzzle so as to avoid changing the old one
    int[] board = new int[9];
    for(int i=0; i<9; i++) {
        board[i]=puzzBoard[i];
    }
    int temp = board[n1];
    board[n1] = puzzBoard[n2];
    board[n2] = temp;
    //add s successor
    eps.add(new EpuzzleState(board));
}
```

Figure 4: changePosition method in BFS

Figure 4: When invoking changePosition, it will change the position of n2 and n1 in the array. What's more, after data position changes, the array with these data will be added in a new EpuzzleState class and it will be stored in the ArrayList.

```
public ArrayList<SearchState> getSuccessors(Search searcher) {


    getSpace(puzzBoard);

    ArrayList<EpuzzleState> zplist = new ArrayList<EpuzzleState>();
    ArrayList<SearchState> slist = new ArrayList<SearchState>();

    if (space != 0 && space != 3 && space != 6) {
        changePosition(space-1, space, zplist);
    }

    if (space != 6 && space != 7 && space != 8) {
        changePosition(space+3, space, zplist);
    }

    if (space != 0 && space != 1 && space != 2) {
        changePosition(space-3, space, zplist);
    }

    if (space != 2 && space != 5 && space != 8) {
        changePosition(space+1, space, zplist);
    }

    for (EpuzzleState z : zplist) {
        slist.add((SearchState) z);
    }
    return slist;
}
```

Figure 5: getSuccessors method in BFS

Figure 5: When invoking getSuccessors, it will check where 0 number is and exchange the position of 0 with which position is near 0. For example, if the index of 0 is 2, the first three if-statement will be invoked then the changePosition will be invoked 3 times.

1.2 A* implementation

Compared with BFS, A* implementation has localCost and estRemCost to help determine which successor will be added and the way in which data is stored is two-dimensional array. The core logic of A* implementation is the same with BFS.

1.2.1 Hamming

```
public int Hamming(int[][] arr) {
    int count=0;
    int[][] goal = {{1,2,3},{4,5,6},{7,8,0}};
    for(int i=0; i<size; i++) {
        for(int j=0; j<size; j++) {
            if(arr[i][j]!=goal[i][j] ) {
                count++;
            }
        }
    }
    return count;
}
```

Figure 6: Hamming method in A*

Figure 6: When invoking Hamming, it will provide the number of data which is out of place. It will check each of them. When meeting a number in not a correct position, the output will be added 1.

1.2.2 Manhattan

```
public int Manhattan(int[][] arr) {
    int output=0;
    for(int i=0; i<size; i++) {
        for(int j=0; j<size; j++) {
            switch(arr[i][j]) {
            case 1:
                output+=Math.abs(i+j);
            case 2,4:
                output+=Math.abs(i+j-1);
            case 3,5,7:
                output+=Math.abs(i+j-2);
            case 6,8:
                output+=Math.abs(i+j-3);
            case 0:
                output+=Math.abs(i+j-4);
            }
        }
    }
    return output;
}
```

Figure 7: Manhattan method in A*

Figure 7: When invoking Manhattan, it will take i and j as coordinates and computer the distance in the way of coordinates. For example,if the

index of 3 is: i=1, j=0, the third case will be invoked. The formula can be: distance = $|i-0|+|j-2|.In programming, it should be Math.abs(i+j-2)$.

2 Results of assessing efficiency for the two search algorithms

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.

Table 1 shows the result of different algorithems(BFS A*). The data example is provided by Handbook and EpuzzleGen.

	BFS	A*(Hamming)	A*(Manhattan)
P1	0.26666668	1	0.4
P2	0.125	0.5844981	0.33208202
P3	0.035714287	0.5	0.13253012
EpuzzleGen6	6.755159E-4	0.0018325346	0.0011671231
EpuzzleGen7	0.0019066846	0.0058242595	0.0049956557
EpuzzleGen8	0.005919662	8.3925476E-4	5.932019E-4
EpuzzleGen9	3.6538023E-4	0.0038129974	0.0022267404
EpuzzleGen10	3.619418E-4	0.005868844	0.0025305315
EpuzzleGen11	4.1277643E-4	0.0016162825	0.0011610906
EpuzzleGen12	0.0002202195	6.3165679E-5	5.771415E-5

Table 1:

3 Conclusions

In summary, after a great number of experiments, the A* is much more efficient than BFS because A* has loaclCost and estRemCost to determine which kind of successors can be the next one.