Project 1: The Eight Puzzle
Yizhuo
12-Feburary-2018

CS 205. Introduction to Artificial Intelligence
Instructor: Dr Eamonn Keogh

In complete this homework, I consulted…
Eamonn Keogh (2018), *Heuristic Search* [Power Point presentation], *http://www.cs.ucr.edu/~eamonn/205*

cplusplus.com (2018), std::priority_queue, http://www.cplusplus.com/reference/queue/priority_queue/

All the code is original. Unimportant subroutines that are not completely original are…
All subroutines used for priority_queue, to sort and get the nodes who should be expanded next.

Section I:

Code:

I pick C++ as my programming language and there are separate files (main.cpp, puzzle.h, puzzle.cpp, solver.h, solver.cpp and ininfo.h) in this project. The important code is showed as following:

```
int main() {
    int puzzlesize = SIZE * SIZE -1;
    printhead(puzzlesize);
    //get initpuzzle
    int choice;
    cin>>choice;
    bool repeat = true;
    int initpuzzle[SIZE][SIZE];
    while(repeat){
        if(choice == 2){
            getinitpuzzle(initpuzzle);
            repeat = false;
        }
        else if(choice == 1){
            repeat = false;
            memcpy (initpuzzle, defaultpuzzle, sizeof(initpuzzle));
        }else{
            cout<<"please input 1 or 2\n";
            cin>>choice;
        }
    }

    puzzle initNode(initpuzzle);
    solver eSolver;
    //pick the huristic function
    askForHFunction();
    repeat = true;
    cin>>choice;
    while(repeat){
        if(choice == 1){
            eSolver.hFunc = cost;
            repeat = false;
        }
        else if(choice == 2){
            repeat = false;
            eSolver.hFunc = misplaced;
        }
        else if(choice ==3){
            repeat = false;
            eSolver.hFunc = manhattan;
        }
        else{
            cout<<"please input 1, 2 or 3\n";
            cin>>choice;
        }
    }
    eSolver.setinit(initNode);
    //begin the search
    eSolver.search();
    return 0;
}
```

```
void solver::search() {
    nodesnum = 0;
    visited.clear();
    pq.push(init);
    queuesize = pq.size();
    if(isGoal(init)){
        findGoal(init,nodesnum,queuesize);
        return;
    }
    while(1){
        if(pq.empty()){
            cout<<"Failure, expanding "<<nodesnum<<" nodes.\n";
            return;
        }
        puzzle currentNode = pq.top();
        pq.pop();
        visited.push_back(currentNode);
        nodesnum++;

        int x;
        int y;
        puzzle nextNode;
        for(int i = 0; i < SIZE; i++){
            for(int j = 0; j < SIZE; j++){
                if(currentNode.board[i][j] == 0){
                    x = i;
                    y = j;
                    break;
                }
            }
        }
        if(x != 0){ //move left
            memcpy(nextNode.board, currentNode.board, sizeof(currentNode.board));
            nextNode.setBoard(x, y, currentNode.board[x-1][y]);
            nextNode.setBoard(x-1, y, 0);
            nextNode.setg(currentNode.getg()+1);
            nextNode.seth(hFunc(nextNode.board, goal.board));
            nextNode.setdepth(currentNode.getdepth() + 1);

            if(isGoal(nextNode)){
                findGoal(nextNode,nodesnum,queuesize);
                return;
            }
            //check if the node has been visited or is already in queue
            if(!hasVisited(nextNode, visited) && (!inqueue(nextNode, pq))){
                pq.push(nextNode);
            }
        }
        if(x != SIZE - 1){ move right
        }
        if(y != 0){
            //move up
        }
        if(y != SIZE - 1){
            //move down
        }
        queuesize = pq.size()>queuesize? pq.size():queuesize;
    }
}
```

Test Case:
A trace of the Manhattan distance A* on the following initial state:
1 2 3
4 8 0
7 6 5
Welcome to Yizhuo's 8-puzzle solver.
Type “1” to use a default puzzle, or “2” to enter your own puzzle.
1
Enter your choice of algorithm
1. Uniform Cost Search
2. A* with the Misplaced Tile heuristic.
3. A* with the Manhattan distance heuristic.3
The best state to expand with a g(n) = 0 and h(n) = 5 is...
1 2 3
4 8 0
7 6 5

Expanding this node ...
The best state to expand with a g(n) = 1 and h(n) = 4 is...
1 2 3
4 8 5
7 6 0

Expanding this node ...
The best state to expand with a g(n) = 2 and h(n) = 3 is...
1 2 3
4 8 5
7 0 6

Expanding this node ...
The best state to expand with a g(n) = 3 and h(n) = 2 is...
1 2 3
4 0 5
7 8 6

Expanding this node ...
The best state to expand with a g(n) = 4 and h(n) = 1 is...
1 2 3
4 5 0
7 8 6

Expanding this node ...
Goal!
To solve this problem the search algorithm expanded a total of 5 nodes.
The maximum number of nodes in the queue at any one time was 8.
The depth of the goal node was 5

Section II: Algorithm Comparison

In this project, three different algorithms are implemented: Uniform Cost Search, A* with the misplaced tile heuristic and A* with the Manhattan distance heuristic. From the project description, the total expanded nodes and the maximum number of nodes in queue are required as a part of output. The time complexity can be indicated by expanded nodes while the maximum number of nodes in queue is relevant to the space complexity.

I design an experiment to compare these three algorithms. From slides, the diameter of 8-puzzle is 31, therefore, I generate 10 puzzles in different depth ($3k + 1, 0 \leq k \leq 10$). By using them as the input, I compare the performance of the three algorithms.
Puzzles generated:
1 2 3
4 5 0
7 8 6 (depth = 1)

0 1 2
4 5 3
7 8 6 (depth = 4)

4 1 2
7 5 3
8 0 6 (depth = 7)

0 4 2
7 1 3
8 5 6 (depth = 10)

7 4 2
1 3 0
8 5 6 (depth = 13)

0 7 4
1 3 2
8 5 6 (depth = 16)

1 7 4
8 3 2
5 0 6 (depth = 19)

0 1 7
8 3 4
5 6 2 (depth = 22)

2 8 7
0 6 5
3 1 4 (depth = 25)

8 6 7
2 0 5
3 1 4 (depth = 28)

8 6 7
2 5 4
3 0 1 (depth = 31)

The experiment result:

Table 1: Nodes Expanded

Problem Depth	1	4	7	10	13	16	19	22	25	28	31
Uniform Cost Search	1	9	85	314	1562	6097	24788	69282	132672	178236	181362
A* (Misplaced tile)	1	4	7	27	148	514	2052	7016	24081	63960	124522
A*(Manhattan)	1	4	7	13	47	82	175	447	712	1783	6860

Figure 1: Total Node Expanded

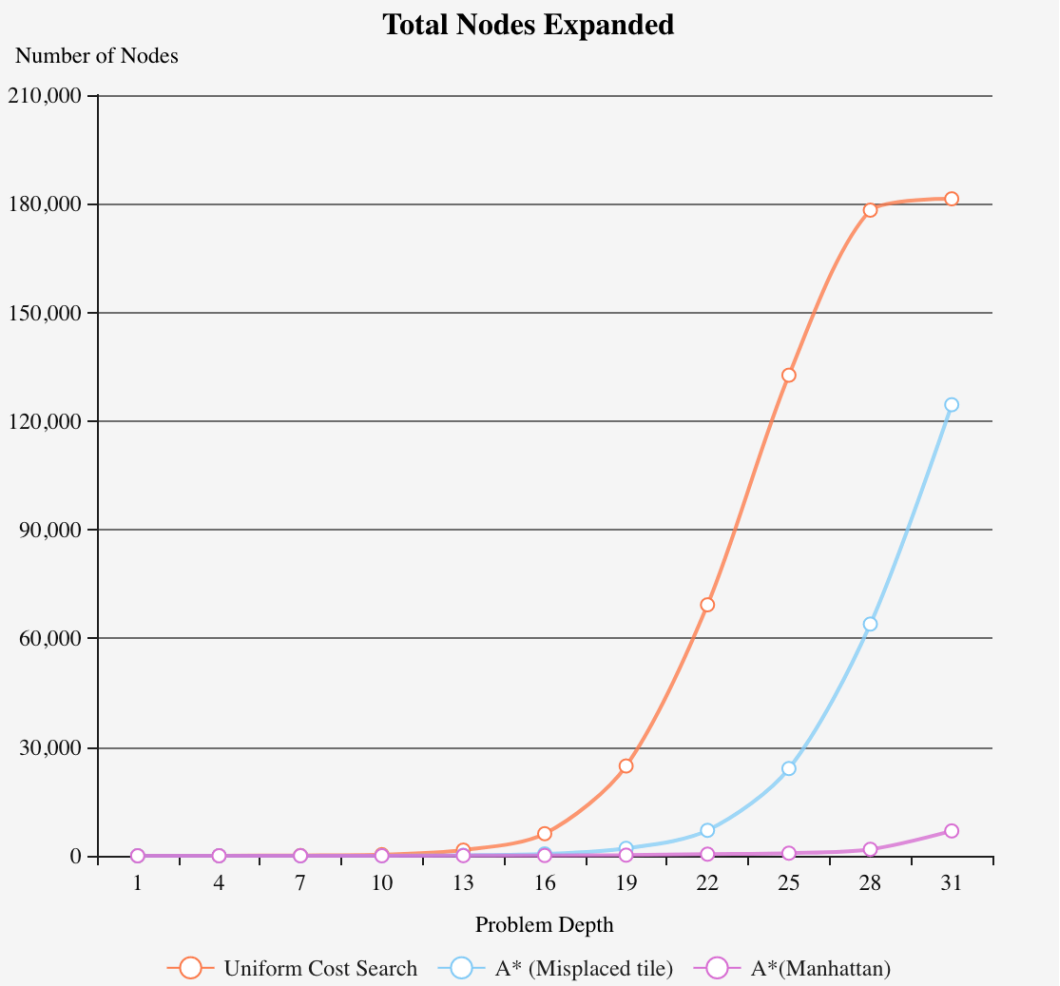
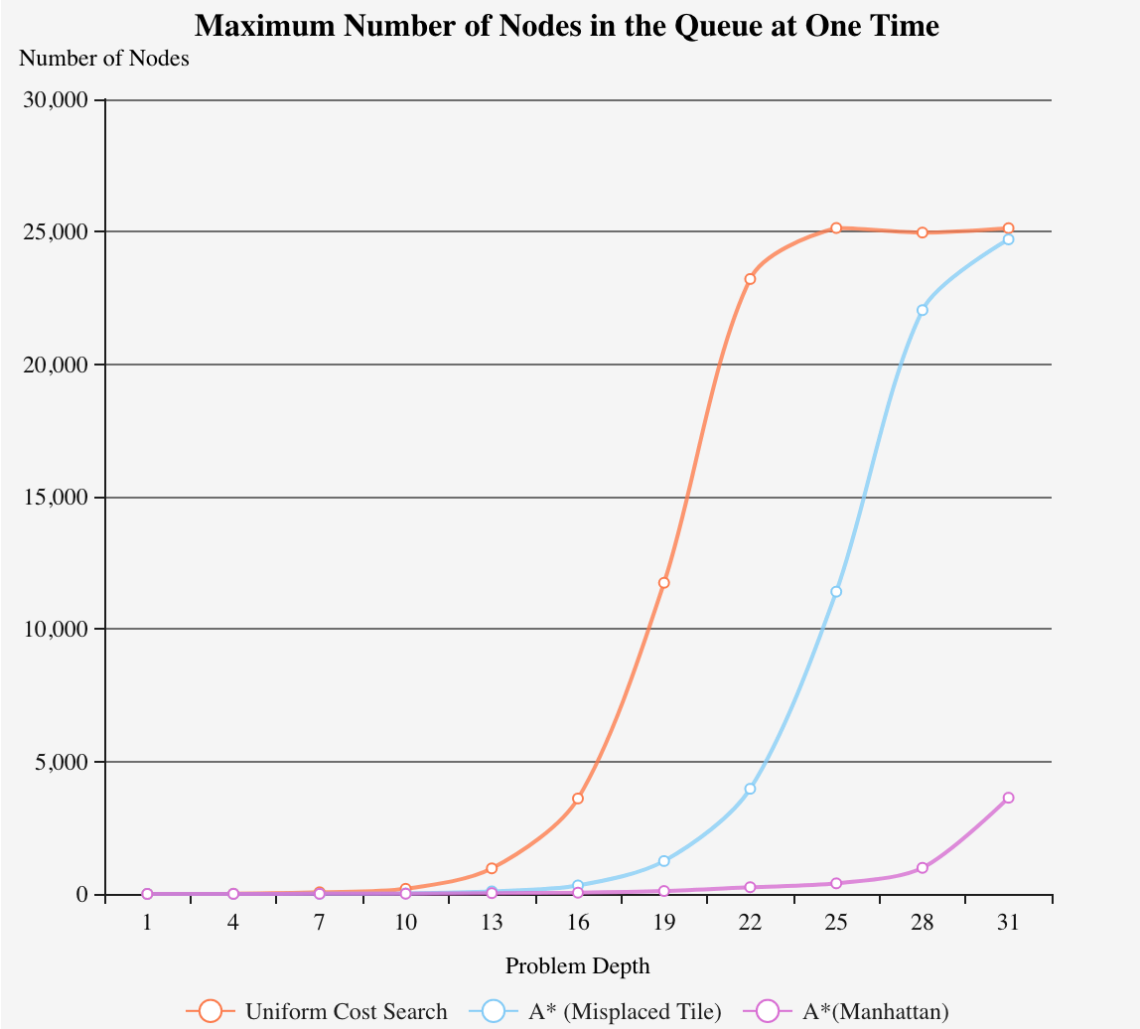


Table 2: Maximum Queue Size

Problem Depth	1	4	7	10	13	16	19	22	25	28	31
Uniform Cost Search	1	9	64	194	965	3603	11747	23216	25142	24969	25135
A* (Misplaced Tile)	1	3	5	19	96	321	1242	3970	11412	22040	24713
A*(Manhattan)	1	3	5	9	30	47	110	254	400	985	3632

Figure 2: Maximum Queue Size



From the previous result, A* with heuristics works better than Uniform Cost Search. To compare the two heuristics, the puzzles whose problem depth from 1 to 10 are used as test cases.

1 2 3
4 5 0
7 8 6 (depth = 1)

1 2 0
4 5 3
7 8 6 (depth = 2)

1 0 2
4 5 3
7 8 6 (depth = 3)

0 1 2
4 5 3
7 8 6 (depth = 4)

4 1 2
0 5 3
7 8 6 (depth = 5)

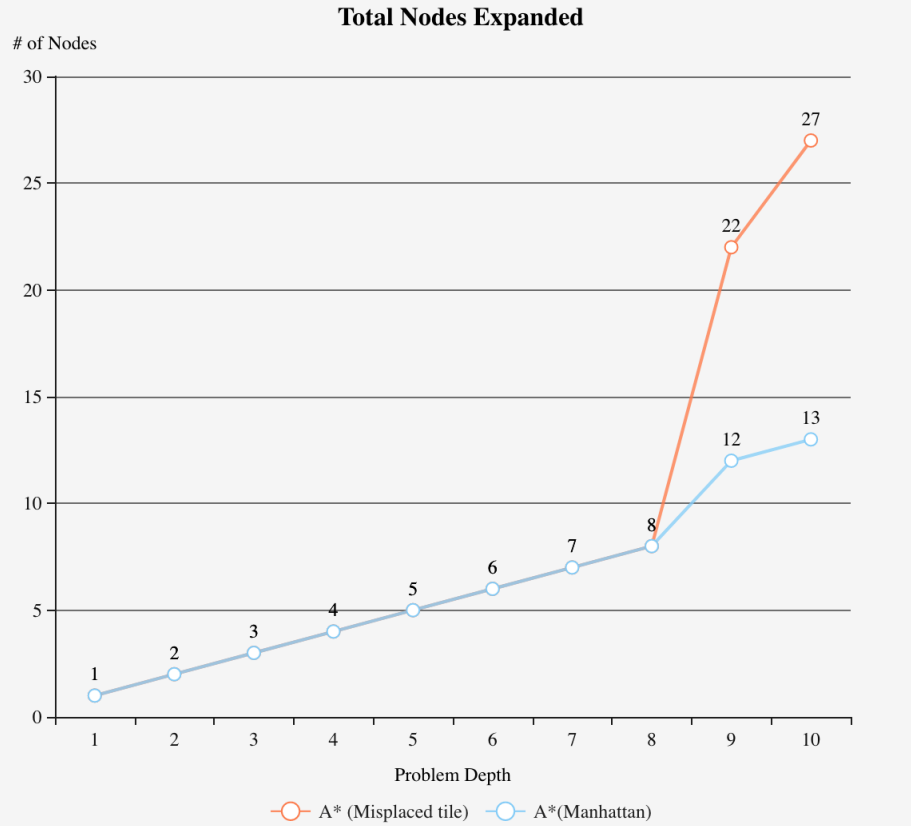
4 1 2
7 5 3
0 8 6 (depth = 6)

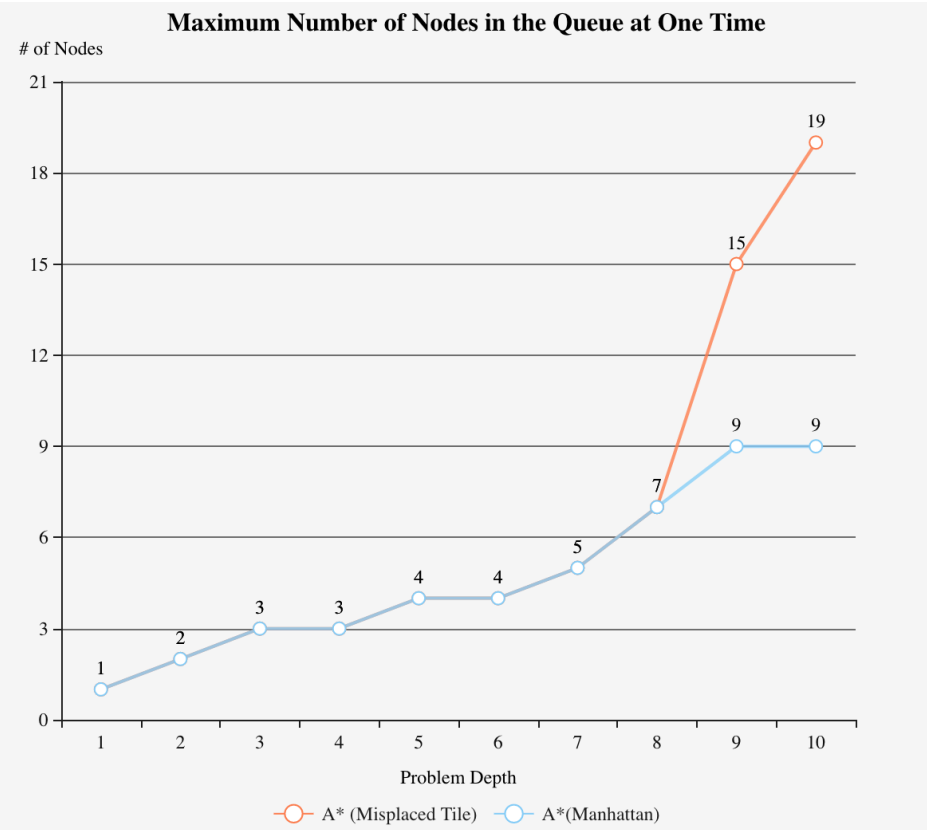
4 1 2
7 5 3
8 0 6 (depth = 7)

4 1 2
7 0 3
8 5 6 (depth = 8)

4 0 2
7 1 3
8 5 6 (depth = 9)

0 4 2
7 1 3
8 5 6 (depth = 10)





Conclusion:
In general, A* with Manhattan heuristics performs better than the other two algorithms while uniform cost search no better than the other two. For some easy puzzles (problem depth is less than 10), the complexity of Uniform Cost Search is acceptable compared to others but the hard puzzles. Here, the uniform cost search is like the breadth first search, both time complexity and space complexity are $O(b^d)$.

Compare A* with Misplaced tiles and Manhattan heuristics, in our cases, they share the competitive complexity before problem depth growing to 9. After that, the A* with Manhattan heuristics works better than A* with Misplaced tiles. The complexity differs a lot when the problem depth grows to 22 or more. We can conclude that different heuristics make the performance A* a huge difference.