Assignment 1 CS205: Introduction to Artificial Intelligence, Dr. Eamonn Keogh

Yuwei Zhang

SID X719520 (concurrent enrollment, from the extension UCP program)

Email yzhan995@ucr.edu

Date May-15-2023

In completing this assignment, I consulted:

- The blind search and heuristic search slides from the lecture
- C++ documentation URL: en.cppreference.com
- Clang git documentation URL: clang.llvm.org/docs/

All important code is original. For simplicity of code implementation, I used built-in function library in C++, include:

- Priority_queue: store the frontier node, sort by f value (fval in struct puzzle)
- Map: eliminate redundant, worse state

Executable code is available at: github.com/yzhan995/205_project1

Outline of this report:

CS205: Assignment 1: The eight-puzzle

Yuwei Zhang, X719520, yzhan995@ucr.edu

Introduction

Sliding tile puzzle, or a sliding puzzle is a combination puzzle that challenges a player to slide pieces along certain routes to establish a certain end-configuration.[1] In our specific task, for example, the 8-puzzle consists of a grid square area, divided into a 3 by 3, with eight of the tiles numbered 1 to 8, and one square left empty. Our task is to restore an arbitrary 8-puzzle state to the standard order: where the tiles arranged in 1 to 8 orderly, and the last grid left with blank. Obviously, it is easy to generalize this problem into bigger version, as long as the 'n-puzzle' n satisfies: $n = m^2 - 1$, where m is a positive integer, $m = 1, 2, \ldots$. The most well-known and presented in lectures include 8, 15, 24 puzzles, which we'll cover some of them in this report.

This assignment is the first project in Dr. Eamonn Keogh's CS205: *Artificial Intelligence* course at the University of California, Riverside during the quarter of Spring 2023. In this project, I program and tested *Uniform Cost Search*, *A* with the Misplaced Tile heuristic* algorithm and *A* with the Manhattan Distance heuristic* algorithm on the 8-puzzle problem. I tested their performance using running time, number of expanded nodes and number of queue frontier nodes as criteria. I coded this on my personal MacBook, using C++ with clang 14.0 compiler (p.s. the compiler and its version are important, I ran my code on g++ 9.4 and it turns out different on expanded nodes when the problem scales up. I thought about this and discussed it with chat-GPT, concluded that it is because the underlying implementation difference of different complier/version. For example, the different computer instruction sequences generated for different computer architectures, would cause difference in searching structure. But, even for large problem sizes, the order of magnitude of the nodes is consistent, and the results remain consistent for the same machine and complier version). Also, I accelerated my search using C++ MAP to compare and storage legal

[1] Quoted from Wikipedia: Sliding puzzle. https://en.wikipedia.org/wiki/Sliding_puzzle

state. This helped a lot in eliminating redundant state and thus the searching time is significantly reduced according to my experiment result. My main code is attached at the last two pages, full code available at github.com/yzhan995/205_project1. According to my experiment result, *A* with Manhattan Distance heuristic* has the best performance in this task.

Comparison of Algorithms

Uniform Cost Search (UCS)

UCS can be viewed as a simplified version of A* algorithm, where h(n) is hardcoded to 0. In this algorithm, every round will only expand nodes with minimum g(n), which is all the nodes with the minimum depth. Note that every expand cost is 1, so the UCS is equivalent to BFS in this specific problem.

A* with the Misplaced Tile heuristic

A* algorithm is the greedy optimized version of UCS, which uses some heuristic algorithms to evaluate the distance between the current state and goal state. Define this function as h(n), so every round just expand node with the minimum f(n) := g(n) + h(n). Misplaced Tile heuristic is a kind of heuristic: uses the number of different grids in the current state and the goal state as the expected distance. In Fig1 for example, there are 4 different grids, so g(n) = 4.

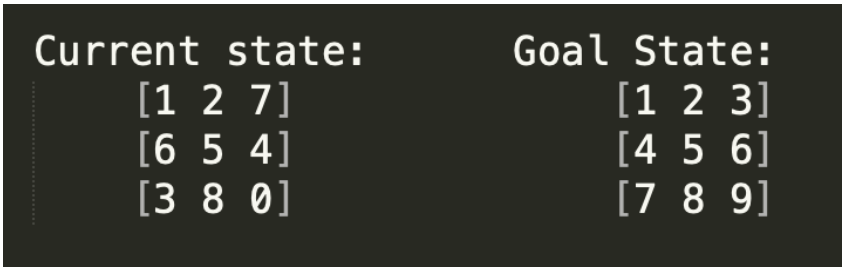

Fig1. An instance, when using Misplaced Tile heuristic, g(n)=4; when using Manhattan distance heuristic, g(n)=12.

A* with Manhattan distance heuristic

Manhattan distance is also a heuristic algorithm, where we take the sum of Manhattan

distance of each corresponding number between the current state and the goal state as the expected distance. For example, in Fig1, grids with number 3, 4, 6, 7 are different from the goal state, so g(n) = 4 + 4 + 2 + 2 = 12.

Implementation and Acceleration

Consider UCS is a special case of A* algorithm where h(n) is hardcoded to 0. The only distinction we should make is in the h(n) for each node. Define data structure puzzle: use matrix[i][j] to store content of the puzzle state, record g, f and h values (as gval, fval and hval) of this state. Note that we should define different calculation method for h when we use different heuristic. See attached fig2.

```
void cal_val(int heuristic = 2) {
// 0 for uniform cost search, 1 for misplaced tile heuristic, 2 for manhattan distance heuristic
    hval = 0;
    if (heuristic == 1) {
        for (int i = 0; i < length; i++)
            for (int j = 0; j < length; j++) { // compared to goal {{1,2,3.}{..}..}
                if (i == length - 1 && j == length - 1) continue; // last is empty is goal
                if (matrix[i][j] != i * length + j + 1) hval++;
            }
    }
    else if (heuristic == 2) {
        for (int i = 0; i < length; i++)
            for (int j = 0; j < length; j++) { // goal_x, goal_y: m[i][j] position in goal
                if (matrix[i][j] == 0) continue;
                int goal_x = (matrix[i][j] - 1) / length;
                int goal_y = matrix[i][j] - goal_x * length - 1;
                hval += abs(goal_x - i) + abs(goal_y - j);
            }
    }
    fval = hval + gval;
}
```

Fig2. Use cal_val(int heuristic) to call f, h and g value of this state. Note heuristic=0,1,2 points to different h calculation method

Here we construct *struct* puzzle:

- Function is_goal(): determine whether this puzzle at the goal state
- Function cal_val(heuristic): calculate h, f and g based on given heuristic, see Fig2
- Function swap_empty(direction): move blank, generate new puzzle state nodes

Define search function A* and expand_nodes(puzzle) in main.cpp:

- Function A_star_search(puzzle): // search from the given puzzle

    Queue.push(puzzle)

    While(queue is not empty) {

        puzzle = queue.pop()

        If (puzzle is not goal puzzle) expand_nodes(puzzle)

Else {print….}

}

- Function expand_nodes(puzzle): expand the given state, move '0' in four ways

Note that how to expand and push new node is important. In A* algorithm, we use prior_queue to store the frontier nodes, and sort it with f value. Obviously we will always search to the repeat puzzle. To avoid that, I use map in C++ standard library, define like this:

- Define map<puzzle, int f> // name 'visit' in my code
- When a new_puzzle is generated:

If new_puzzle exists in map && its f value better

Continue; //ignore it

Else

Queue.push(new_puzzle)

Map[new_puzzle] = new_puzzle_fval // push and update in map

The following experiments show that using map greatly improve the search efficiency and reduced a lot redundant work. P.s. if we want to run without map, just comment out line 35 & 38 in npuzzle.cpp

Experiment and Results

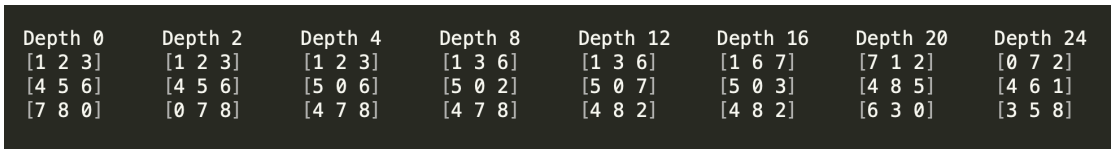

Fig3. The default 8 test case

Fig3 is from Dr. Keogh's default test sample. Depth increases from left to right. I tested the three algorithms on these cases, compared to whether use map to accelerate or not. See Table1 & Table2. Note that 'Expand' represent Expanded nodes, 'Frontier' represents the maximum Frontier nodes. They respectively correlate with *time spent* and *memory needed*, according to Dr. Eammon's definition and understanding.[2] P.s.

[2] Referenced from slides 3_Heuristic Search, page 3

N/A means no result within 10seconds, then manually terminated program.

Table1. Comparation of three algorithms (not using map)

Case	Depth	Uniform Cost Search			A* with the Misplaced Tile heuristic			A* with the Manhattan Distance heuristic		
		Time	Expand	Frontier	Time	Expand	Frontier	Time	Expand	Frontier
1	0	0	1	1	0	1	1	0	1	1
2	2	0	8	14	0	3	4	0	3	4
3	4	0	74	136	0	5	9	0	5	9
4	8	6	3521	6625	0	19	33	0	13	23
5	12	625	587520	1027534	4	1561	2775	0	83	138
6	16	N/A	N/A	N/A	90	87861	157585	4	1370	2345
7	20	N/A	N/A	N/A	1491	1231990	2270415	32	23204	39047
8	24	N/A	N/A	N/A	N/A	N/A	N/A	213	181831	311565

Table2. Comparation of three algorithms (using map)

Case	Depth	Uniform Cost Search			A* with the Misplaced Tile heuristic			A* with the Manhattan Distance heuristic		
		Time	Expand	Frontier	Time	Expand	Frontier	Time	Expand	Frontier
1	0	0	1	1	0	1	1	0	1	1
2	2	0	7	8	0	3	3	0	3	3
3	4	0	32	28	0	5	6	0	5	6
4	8	1	326	193	0	19	16	0	13	12
5	12	8	1887	1198	0	118	83	0	36	27
6	16	45	13025	6563	4	648	402	1	94	62
7	20	230	52579	17785	11	2725	1607	2	424	270
8	24	784	126138	24212	64	17353	8591	5	925	538

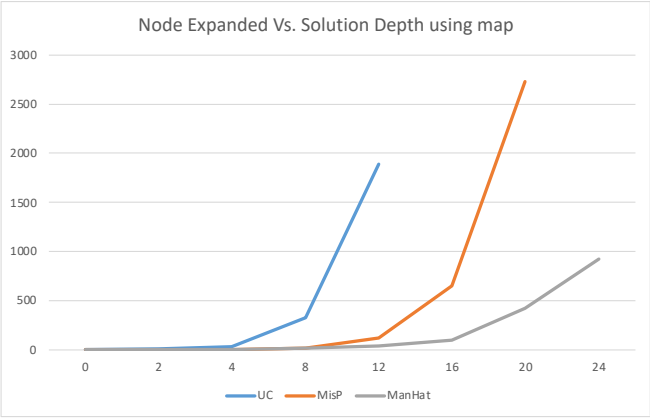

Fig4. Node expanded (using map)

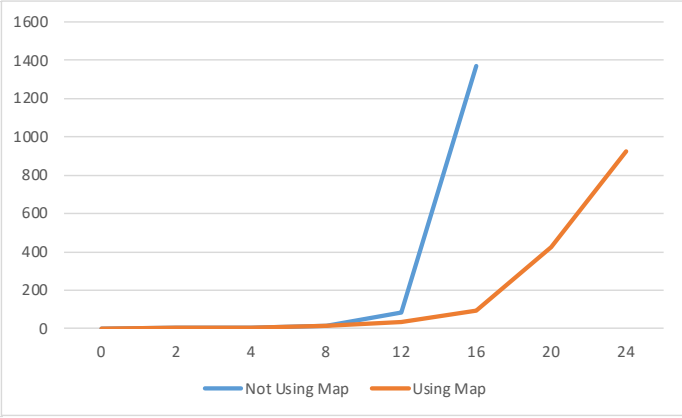

Fig5. Manhattan not using map vs map

It’s easy to see, when faced with larger problem sizes, the efficiency of three heuristics varies widely, especially after using map acceleration. For example, in case 8 (see Fig1) where depth=24, the expanded nodes for Manhattan Distance heuristic after map acceleration is only 925, but the UCS and Misplaced Tile heuristic expanded node are respectively higher than 100k and 10k, exactly an order of magnitude difference in execution time. It is also worth noting that the map did great help with reduce the searching size of problem. See Fig 5.

It can be said that in this problem, the execution complexity and efficiency of the three algorithms are always Manhattan greater than Misplaced Tile heuristic greater than UCS.

Conclusion

In this specific task: 8-puzzle problem, among the three heuristic algorithm, Manhattan Distance heuristic always achieve best performance. It is far superior to the other two in terms of search scale and memory needed. The gap between even more obvious as the depth of the problem increases. Then Misplaced Distance heuristic perform better than Uniform Cost Search. This indicates that the choice of heuristic function will greatly affect the efficiency of A*, how to choose a good heuristic algorithm is the key to determine the efficiency of search.

At the same time as a comparison, taking out redundant and worse states can greatly speed up problem solving (see Fig 5). It shows that the searching process contains a large number of redundant states, and the use of appropriate data structures can effectively avoid it.

The following is a traceback of an 8-puzzle with depth=4 (case 3 in Fig 1), using Manhattan Distance with map acceleration:
Executable code at github.com/yzhan995/205_project1

Enter '0' to use the Uniform Cost Search algorithm, '1' to use the A* with the Misplaced Tile heuristic, '2' to use the A* with the Manhattan Distance heuristic:
2
Reach goal state!
Move Process:
1 2 3
5 0 6
4 7 8
  |
  V
1 2 3
0 5 6
4 7 8
  |
  V
1 2 3
4 5 6
0 7 8
  |
  V
1 2 3
4 5 6
7 0 8
  |
  V
1 2 3
4 5 6
7 8 0
Depth: 4
Number of nodes expanded: 5
Number of maximum frontier node: 6
Time taken by program: 0 milliseconds

The following is a traceback of a 15-puzzle with depth=4, using Manhattan Distance with map acceleration:
p.s. if you want to run n-puzzle, modify const int puzzle_size = n in npuzzle.cpp

➜ npuzzle ./npuzzle

Enter the 1th row: 1 2 3 4
Enter the 2th row: 5 6 7 8

Enter the 3th row: 0 10 11 12
Enter the 4th row: 9 13 14 15
1 2 3 4
5 6 7 8
0 10 11 12
9 13 14 15
Enter '0' to use the Uniform Cost Search algorithm, '1' to use the A* with the Misplaced Tile heuristic, '2' to use the A* with the Manhattan Distance heuristic:
2
Reach goal state!
Move Process:
1 2 3 4
5 6 7 8
0 10 11 12
9 13 14 15
|
V
1 2 3 4
5 6 7 8
9 10 11 12
0 13 14 15
|
V
1 2 3 4
5 6 7 8
9 10 11 12
13 0 14 15
|
V
1 2 3 4
5 6 7 8
9 10 11 12
13 14 0 15
|
V
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 0
Depth: 4
Number of nodes expanded: 5
Number of maximum frontier node: 5
Time taken by program: 0 milliseconds