HW 2

1st Chengkai Wu
Xidian University
Xi'an, China
ckwu1201@163.com

***Abstract*—In this report, I compare the operating efficiency of A* and JPS algorithms in different heuristic functions and with or without Tie Breaker. The best method is the JPS algorithm that uses manhattan distance as the heuristic function. Its running time is 0.208ms and the number of visited nodes is 46.Meanwhile, the path found is only 0.84% longer than the optimal path. When it comes to the empty map, the JPS algorithm using manhattan distance and diagonal distance as heuristic functions run longer time than A* algorithm by 1056.5% and 109.2% respectively. Joining Tie Breaker will reduce the number of nodes visited, but increase the running time varying from method to method.**

***Index Terms*—A*; JPS; Heuristic Function; Tie Breaker**

I. A* Algorithm

Algorithm 1: A* algorithm

Data: Grid Map $\mathbf{M}$, Openlist $\mathbf{O}$, Closedlist $\mathbf{C}$
Input: Start $\mathbf{S}$, Goal $\mathbf{G}$
Output: Path $\mathbf{P}$

Initialize:$\mathbf{O} \leftarrow \varnothing, \mathbf{C} \leftarrow \varnothing$
$\mathbf{O}.insert(f(\mathbf{S}), \mathbf{S})$
while $\mathbf{O} \neq \varnothing$ **do**
  $N_{current} \leftarrow *\mathbf{O}.begin()$
  $\mathbf{O}.erase(f(N_{current}), N_{current})$
  $\mathbf{C}.insert(f(N_{current}), N_{current})$
  if $N_{current} == \mathbf{G}$ **then**
    break
  end
  $N \leftarrow getSucc(N_{current}, \mathbf{M})$
  for N_i *in* N **do**
    if $N_i \notin \mathbf{O} \cup \boldsymbol{C}$ **then**
      $N_i.g \leftarrow g(N_{current}, N_i)$
      $N_i.h \leftarrow h(N_i)$
      $\mathbf{O}.insert(f(N_i), N_i)$
    else
      if $N_i \in \mathbf{O}$ & $\mathbf{g(N_{current}, N_i) < N_i.g}$ **then**
        $\mathbf{O}.erase(f(N_i), N_i)$
        $N_i.g \leftarrow g(N_{current}, N_i)$
        $\mathbf{O}.insert(f(N_i), N_i)$
      end
    end
  end
end
if $N_{current} == \mathbf{G}$ **then**
  $\mathbf{P} \leftarrow \mathbf{getPath(N_{current}, S)}$
end

II. Heuristic Function

A heuristic function is an estimated distance from a node N(x,y,z) to the goal G(x,y,z), which can be added to the accumulated cost to estimate the length of the path going through N, and thus improve the efficiency of search. The following is three different types of heuristic function.

A. Dijkstra

$$D = 0 \tag{1}$$

B. Manhattan

$$D_M = \sum_{i=1}^{3} |N_i - G_i| \tag{2}$$

C. Euclidean

$$D_E = \sqrt{\sum_{i=1}^{3} (N_i - G_i)^2} \tag{3}$$

D. Diagonal

Let

$$d_i = |N_i - G_i|, i = 1, 2, 3.$$

Sort d_i and make sure that $d_1 \leq d_2 \leq d_3$, then

$$D_D = \sqrt{3}d_1 + \sqrt{2}(d_2 - d_1) + (d_3 - d_2). \tag{4}$$

III. Tie Breaker

For the nodes with the same minimum f score in the open list, it is not necessary to visited all of them. So when there are more than a node with the same minimum f score, the program will compare their h score and choice the node with minimum h score to visit.

IV. EXPERIMENTAL RESULTS

I run all methods under both a complex map with many obstacles and an open map. In order to get the same goal in different ways, I wrote a node that continuously sends the goal. For each method, the program will run 10 times, and the program will output the average of the results. The target is set at (4.78851, 3.76074, 0.5). Fig 1 and Fig 2 respectively show the shortest path obtained by the program under complex and empty maps. Table I shows the average results of different methods.

TABLE I
RESULT

Method	Running Time(ms)	Length(m)	Nodes
Dijkstra	36.470792	6.867657	21471
A* Manhattan	0.305708	7.414068	57
A* Euclidean	6.683100	6.867657	1808
A* Diagonal	1.749055	6.867657	592
JPS Dijkstra	30.415485	6.886933	11115
JPS Manhattan	0.208570	6.925483	46
JPS Euclidean	2.988392	6.867657	888
JPS Diagonal	0.716259	6.867657	283
Dijkstra Tie Breaker	41.234491	6.867657	21471
A* Manhattan Tie Breaker	0.369250	7.052619	50
A* Euclidean Tie Breaker	6.553470	6.867657	1803
A* Diagonal Tie Breaker	2.477988	6.867657	523
Dijkstra*	33.183496	6.281871	23418
A* Manhattan*	0.228987	6.281871	23
A* Euclidean*	4.062847	6.281871	917
A* Diagonal*	1.226371	6.281871	328
JPS Dijkstra*	5.983003	6.281871	199
JPS Manhattan*	2.660851	6.925483	3
JPS Euclidean*	1.621907	6.867657	4
JPS Diagonal*	2.565293	6.281871	3
Dijkstra Tie Breaker*	48.493228	6.281871	23418
A* Manhattan Tie Breaker*	0.247523	6.281871	23
A* Euclidean Tie Breaker*	3.813242	6.281871	910
A* Diagonal Tie Breaker*	2.147585	6.281871	325

the '*' at the end of methods indicates the result under the empty map

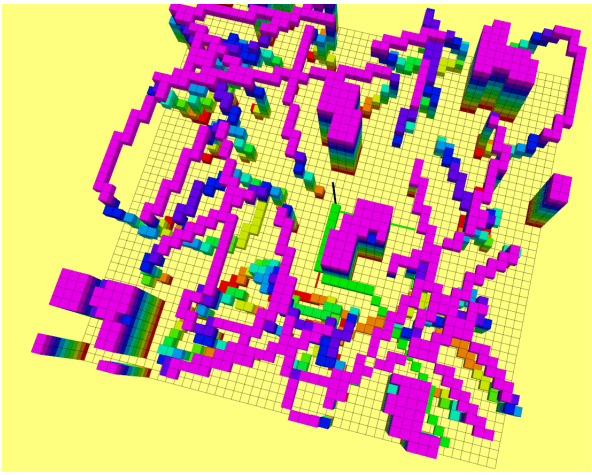

Fig. 1. Complex map and a path from the start point to the goal point generated by the program

It can be seen from the results in Table I that under a complex map, for all heuristic functions, the JPS algorithm has improved running time, shortest path, and number of access nodes compared to the A* algorithm. The best method is the JPS algorithm that uses manhattan distance as the heuristic function. Its running time is 0.208ms and the number of visited nodes is 46, which respectively is one thrid and one seventh of the results obtained by the second best method. What's more, the path found is only 0.84% longer than the optimal path. From the perspective of heuristic function, when manhattan distance is used as the heuristic function of the algorithm, the running time is the shortest. But the path obtained is not

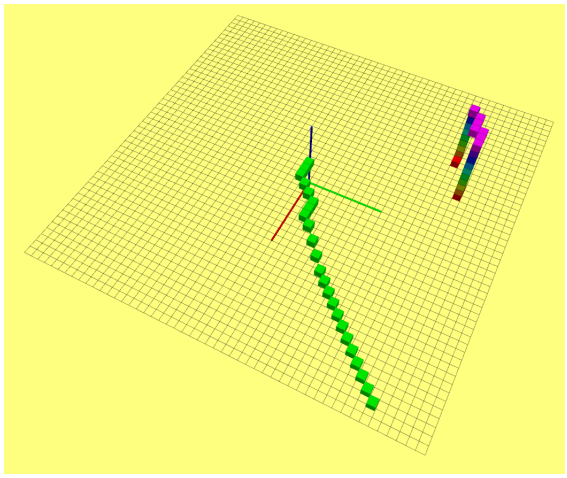

Fig. 2. Empty map and a path from the start point to the goal point generated by the program

optimal,which is because the distance from the current node to the target node estimated by the Manhattan distance will be longer than the actual optimal distance, resulting of the heuristic function is not admissible.

In an empty map, although the JPS algorithm greatly reduces the number of node visits, when the algorithm uses manhattan distance and diagonal distance as heuristic functions, the running time increases by 1056.5% and 109.2%, respectively. This is because in an empty map, the JPS algorithm will search for too many unnecessary nodes during the jump process, resulting in the JPS algorithm's longer running time than the A* algorithm.

Whether it is a complex or empty map, for all methods, joining Tie Breaker will reduce the number of visited nodes slightly, but the running time will increase to varying degrees.

V. Problems I met

A. Problem I

When I set the goal at the edge of the map, the system reported errors that Segmentation fault (Address not mapped to object [(nil)]) as it is showed in Fig. 3. Finally, I find that when the program get the neighbors, the feasible index ranges from 0 to GL_SIZE-1 rather than from 0 to GL_SIZE.

```
std::allocator<void> > const&)
#2    Object "/home/ck1201/workspace/FASTLAB/MotionPlanning/devel/lib/grid_path_
searcher/demo_node", at 0x55d138606566, in pathFinding(Eigen::Matrix<double, 3,
1, 0, 3, 1>, Eigen::Matrix<double, 3, 1, 0, 3, 1>)
#1    Object "/home/ck1201/workspace/FASTLAB/MotionPlanning/devel/lib/grid_path_
searcher/demo_node", at 0x55d138619d92, in AstarPathFinder::AstarGraphSearch(Eig
en::Matrix<double, 3, 1, 0, 3, 1>, Eigen::Matrix<double, 3, 1, 0, 3, 1>)
#0    Object "/home/ck1201/workspace/FASTLAB/MotionPlanning/devel/lib/grid_path_
searcher/demo_node", at 0x55d138619583, in AstarPathFinder::AstarGetSucc(GridNod
e*, std::vector<GridNode*, std::allocator<GridNode*> >&, std::vector<double, std
::allocator<double> >&)
Segmentation fault (Address not mapped to object [(nil)])
```

Fig. 3. Errors: Segmentation fault (Address not mapped to object [(nil)])