HW 2

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

***Abstract*—In this report, I compare the operation efficiency of A* algorithm with different heuristic functions, with and without Tie Breaker. It turns out that A* with diagonal heuristic functions gets the shortest path in a complex map at the cost of visiting 592 nodes and consuming 1.749ms, which is far better than other methods. When it comes to a empty map, A* with manhattan heuristic functions is the best. Adding tie breaker to the program will only slightly reduce the number of nodes visited, but it will greatly increase the running time.**

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

I. A* Algorithm

Algorithm 1: A*

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. Manhattan

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

B. Euclidean

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

C. Diagonal

Let

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

Make sure that $d_1 \leq d_2 \leq d_3$, and

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

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

In order to generate the same experimental environment, I set the random seed as 1 when generating random map. What's more, I write a new node to publish a goal so that the searcher will receive the same goal. For each method, the program will run 10 times and then output the average of the results. I set the goal at (4.78851,3.76074,0.5) and create a complex map and an empty maps to test different methods.Fig 1 and Fig 2 shows the complex and empty maps. Table I shows the results of different methods.

From the first four lines of Table I, we can find that although A* Manhattan has the shortest running time, its path length is not optimal because Manhattan distance always longer than the realistic shortest path. Compared with A* Euclidean, A* Diagonal runs shorter times and visits less nodes as a result of

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

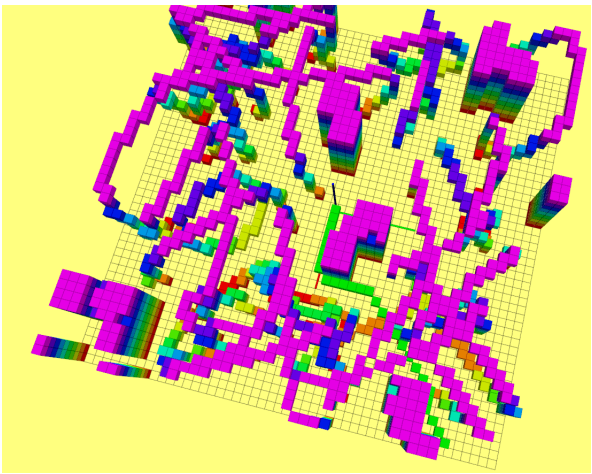

Fig. 1. Complex map and a path from the start point to the goal point generated by the program

that Diagonal distance is closer to the realistic shortest distance from the start point to the goal point than Euclidean distance. Namely, Diagonal distance is more accurate in estimating the realistic shortest path.

For the methods with Tie Breaker, it shows that the program runs longer time but visits a little fewer nodes. A* Manhattan with Tie Breaker obtains a better path than A* Manhattan.

For the map with less obstacles, it turns out that A* Manhattan finds the shortest path at at the cost of both least running time and visited nodes.

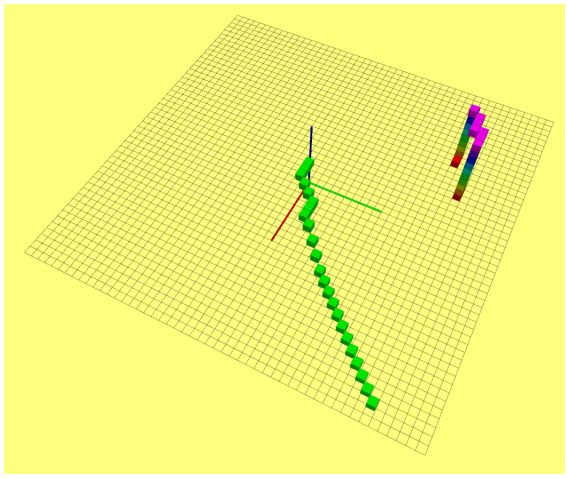

Fig. 2. Empty map and a path from the start point to the goal point generated by the program

V. Problems I meet

A. Problem I

When I set the goal at the edge of the map, the system reports 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 0 to GL_SIZE.

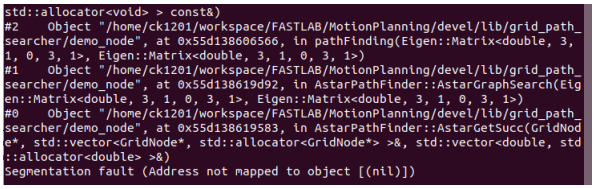

Fig. 3. Errors: Segmentation fault (Address not mapped to object [(nil)])

B. Problem II

When there are more than one node in the open list that has the same smallest f score, I create a multimap type container to store these nodes. Before the program push a node into the container, it should judge that if the iterator is equal to the last iterator of the open list first, otherwise the program may overflow.

```
while (it != openSet.end())
{
    if(tempPtr->fScore != fmin){
        break;
    }
```

Fig. 4. Judge that if the iterator is equal to the last iterator of the open list first and then judge that if the node has the same f score.