Motion Planning

Hisu-Wen Yen A59010599 hsyen@ucsd.edu

I. Introduction

Motion planning is a process for a robot to move from initial state to goal state, and we can also add extra condition on motion planning to make robot move in a way we want. In robotics, motion planning is very important because improper motion planning won't let robot complete its task. For example, autonomous car requires proper motion planning to park automatically without hitting obstacles.

In this problem, we have robot, target, and a few obstacles on the 2D grid map. We need to design a path for robot to reach target. However, target will make a move to keep away from robot for every two seconds. Therefore, we need to find new path for robot after target moves.

To find the shortest path in complex environment, we can use A* algorithm. However, 2 seconds constrain makes A* not suitable for the task. Therefore, I use real-time adaptive A* to make sure robot can always make next move within two seconds.

II. Problem Formulation

In this problem, our robot has deterministic move to next location. Therefore, we can transform the problem into Deterministic Shortest Path (DSP) problem.

We can according to given map information to build a graph with vertices V, edges ε, edge weights c, start node $s \in V$ and terminal node $\tau \in V$. c_{ij} is the cost from vertex i to vertex j.

Vertex is defined as one pixel on the map, and we use coordinates of x axis and y axis to identify each vertex.

$$v_i \begin{cases} x_i \\ y_i \end{cases}, \quad \forall v_i \in V$$

$$x_i = x\ coordinate\ of\ vertex\ i$$

$$y_i = y\ coordinate\ of\ vertex\ i$$

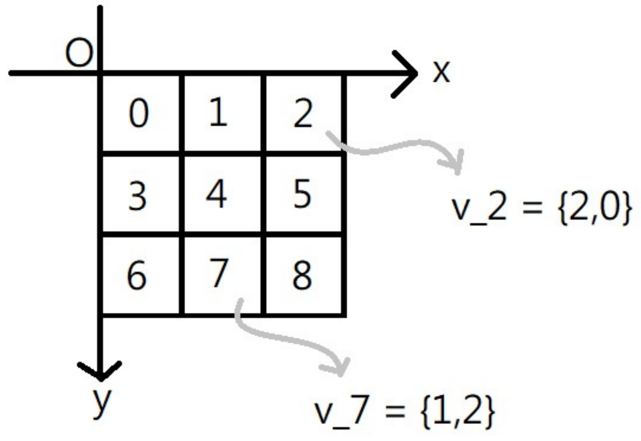


Fig1 Illustration of vertices construction

Vertices are directly extracted from given 2D map. Therefore, edge connections and weights are Euclidean distance between two adjacent vertices.

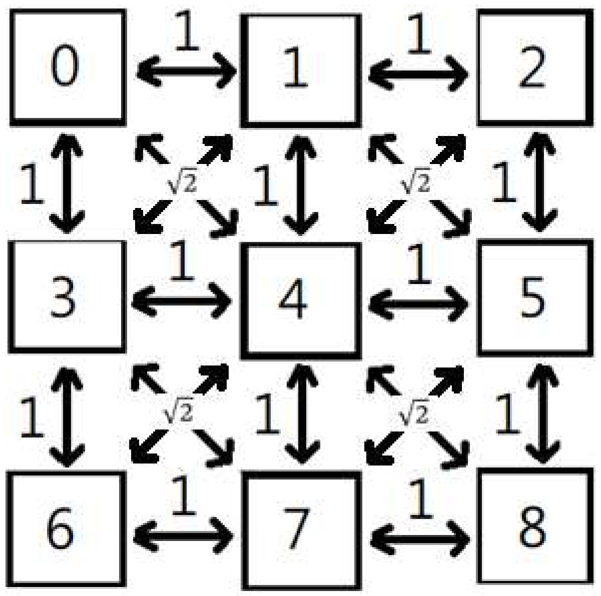


Fig2 Illustration of edges and edges weights

When a vertex is denoted as obstacle in given map, we simply make it lose all connections with other vertices to make it unreachable since out robot can't pass through a wall.

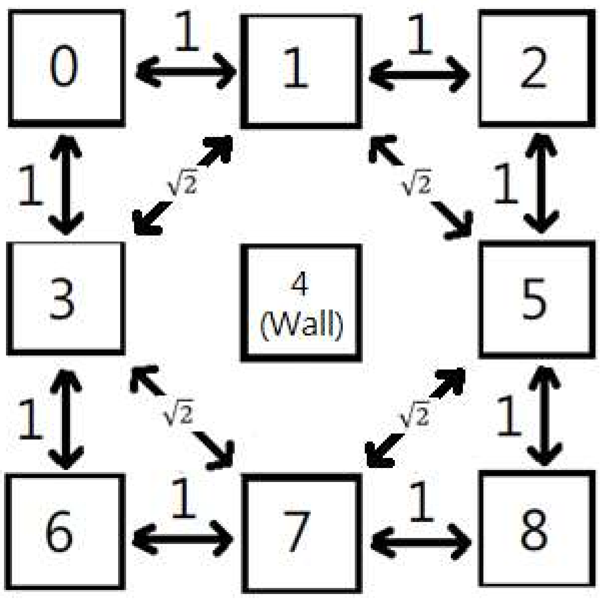


Fig3 Illustration of edges and edges weights if v_4 is wall

We can use graph above to find a path from $s \in V$ to $\tau \in V$: $P_{s,\tau} = \{i_{1:q} | i_k \in V, i_1 = s, i_q = \tau\}$, and we can define the path cost as $J^{i_{1:q}} = \sum_{k=1}^{q-1} c_{i_k,i_{k+1}}$.

Our problem becomes find a path that has min cost from node s to node τ.

$$dist(s,\tau) = min_{i_{1:q} \in P_{s,\tau}} J^{i_{1:q}}$$

$$i^*_{1:q} = argmin_{i_{1:q} \in P_{s,\tau}} J^{i_{1:q}}$$

For convenience, denote g_i as path cost from node s to node i.

$$g_i = dist(s,i)$$
$$g_i + c_{i,j} = dist(s,j) = g_j$$

III. Thectival Approches

A. A*

We can use A* algorithm to solve the DSP. A* will directly return shortest path from s to τ. However, in this problem our target will move after our robot moves. Therefore, we need to recompute shortest path by A* again until robot catches target.

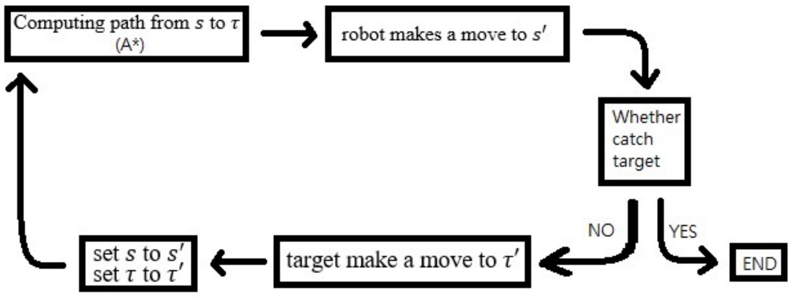

Computing path from s to τ (A*)
robot makes a move to s'
Whether catch target
NO
YES
END
target make a move to τ'
set s to s'
set τ to τ'

Fig4 Process for applying A*

However, in this problem robot and target don't move synchronously. A* will compute the complete path from s to τ, it's possible that it takes more than 2 seconds to find path. In this case our robot will never catch target because one movement is always less than two movements. Therefore, in large scale map A* isn't suitable.

B. Real-Time adaptive A*

To make robot decide next move within 2 sec, I use RTAA*. The main difference is that RTAA* only expand certain amount of cells. Even though this method can't find the optimal path, it still can find a path from s to τ, and it can decide next move faster than A*. That is why RTAA* can work on larger scale map. Set number of cells to expand as n.

To do RTAA* we need a few steps:

1) Define heuristic function for every vertex

In this problem, our robot can move diagonally. Therefore, I choose Euclidean distance to maintain admissible and consistent.

$$h_i = \sqrt{(x_i - x_\tau)^2 + (y_i - y_\tau)^2}$$

2) Expand nodes

Here we do the same thing as A*. However, we stop expanding nodes once the number of cells in $CLOSE$ set is larger than n.

```
Pseudo Code: A* with $n$ expansion
OPEN← $\{s\}$, $CLOSE = \{\}$ $g_s = 0, h_\tau = 0$
While True:
  $Sort(OPEN)$
  Remove first element $i$ from $OPEN$
  $CLOSE = CLOSE \cup \{i\}$
  for $j \in$ Children $(i)$:
    if $(g_i + c_{ij}) < g_j$ and $(g_i + c_{ij}) < g_\tau$:
      $g_j = g_i + c_{ij}$
      $OPEN = OPEN \cup \{j\}$
  if $\tau$ in $CLOSE$ or $len(CLOSE)$ >= n:
    $break$
```

* $Sort(L)$ is a function that reorder element i in list L by their f_i value from small to large. ($f_i = g_i + h_i$)

3) Update heuristic in $OPEN$

When target moves, we can treat it as a change in environment. Therefore, we need to update heuristic for each vertex, and it is time consuming for large map. A way to reduce the time is to only update important vertices. Important vertices here are vertices in $OPEN$ list because they contain potential path to τ.

Pseudo Code: Update heuristic in $OPEN$

```
for $i$ in $OPEN$:
    e_dist = $\sqrt{(x_i - x_\tau)^2 + (y_i - y_\tau)^2}$
    if $flag_i == False$:
        $h_i = e\_dist$
    else:
        $h_i = \max(h_i, \text{e\_dist})$
```

* flag is used to make computer memorize which vertex has been updated before. If the vertex is not updated before, we can directly update it with Euclidean distance with τ because it isn't adaptive to the environment now.

Proof of admissibility:

Euclidean distance is admissible and consistent.

Old h is also admissible and consistent.

Therefore, maximum of both value will be admissible and consistent, too.

4) Update heuristic in $CLOSE$

In this step, we update heuristic in $CLOSE$ to prepare the environment for next iteration.

$$new\ h_i = f_{j*} - g_i$$
$$j^* = argmin_j g_j + h_j$$
$$\text{Where } j \in OPEN$$

Proof of admissibility:

$$\text{dist}(i,\tau) \geq \text{dist}(s,\tau) - g_i \geq f_{j*} - g_i$$

Pseudo Code: Update heuristic in $CLOSE$

```
$Sort(OPEN)$
$j^*$= first element in $OPEN$
$f_{j*} = g_j + h_j$
for $i$ in $CLOSE$:
    $h_i = f_{j*} - g_i$
    $flag_i = True$
```

5) Move the robot

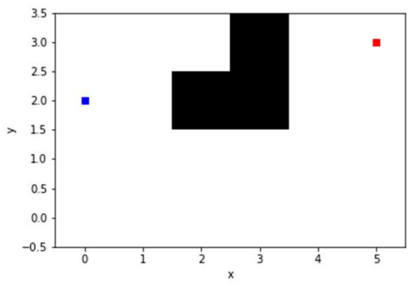
Move the robot with one step on the path to j^*, and rest the g value to infinity for all nodes to prepare for next iteration. Then repeat the process from step 2) until robot catch target.

IV. Results

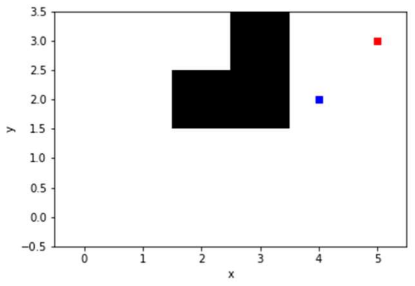
To have a better visualization on the results, please refer to the gif animation in zip file.

A. Test on different map

MAP 0



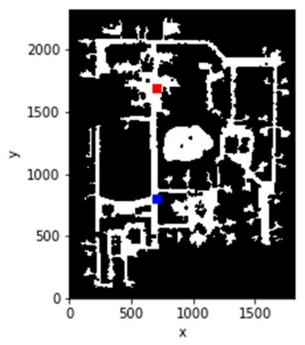
t = 0



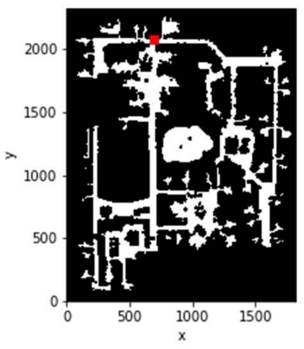
t = 4

$n = 4, number\ of\ steps = 4, caught = True$

MAP 1

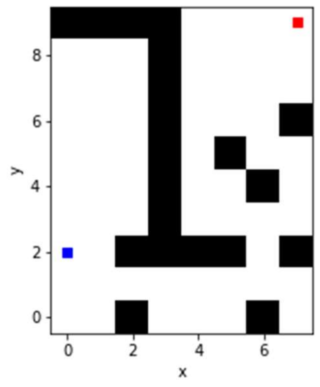


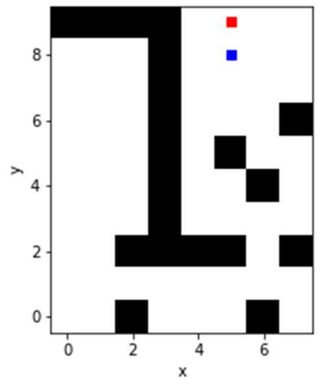
t = 0


t = 1279

$n = 7, number\ of\ steps = 1279, caught = True$

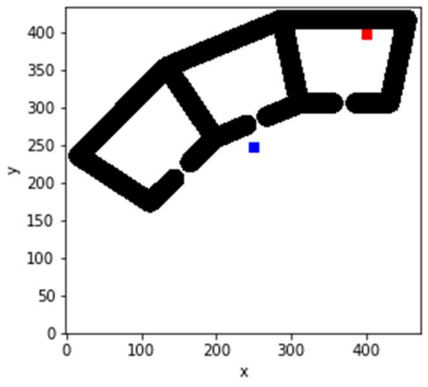
MAP 2

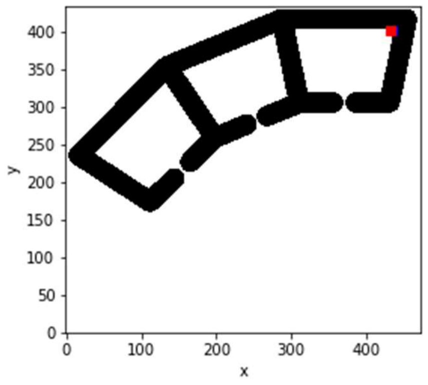

t = 0


t = 28

$n = 9, number\ of\ steps = 28, caught = True$

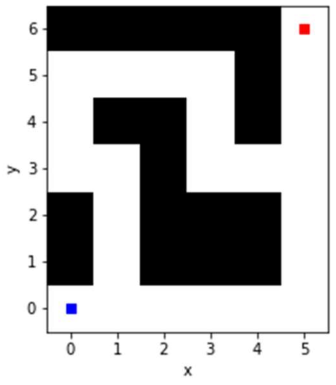
MAP 3

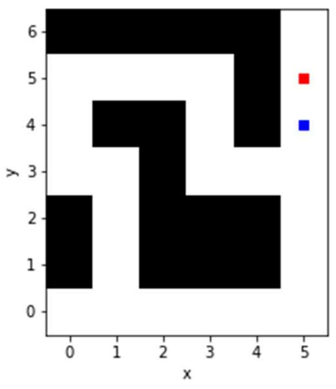

t = 0


t = 223

$n = 2, number\ of\ steps = 223, caught = True$

MAP 4

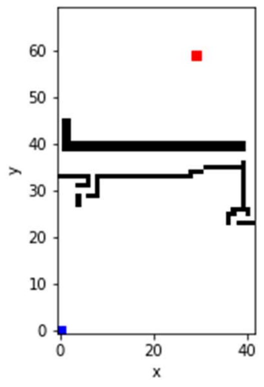

t = 0



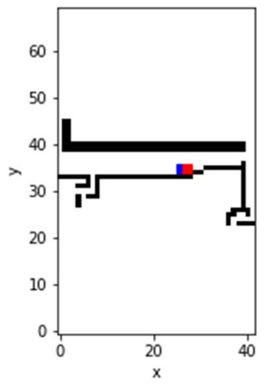
t = 9

$n = 3, number\ of\ steps = 9, caught = True$

MAP 5



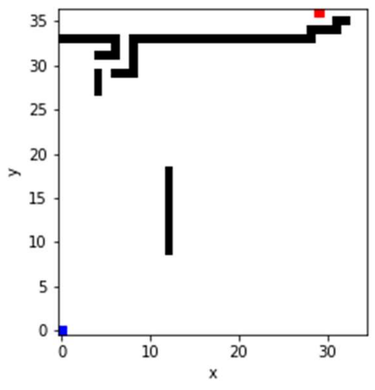
t = 0



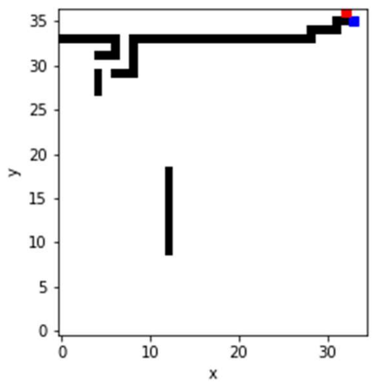
t = 172

$n = 4, number\ of\ steps = 172, caught = True$

MAP 6



t = 0



t = 41

$n = 4, number\ of\ steps = 41, caught = True$

MAP 7

Fail! It can’t catch target in 20000 iterations.

Discussion:

Except for MAP 7, all other map can sucessfully complete the task. It verifies that our robot can always make a move within 2 seconds. Otherwise, the target will move further than robot.

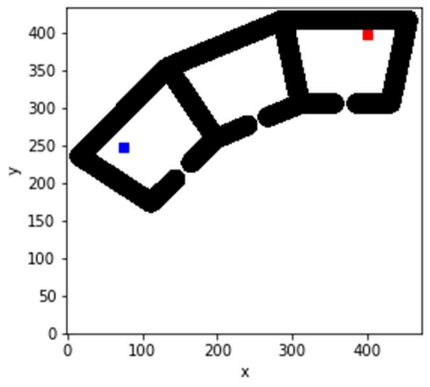
The reason for MAP 7 fail is because MAP 7 is too big, it needs more than 20000 iterations for robot to catch target.

B. Test on diffrerent initial state

MAP 1b

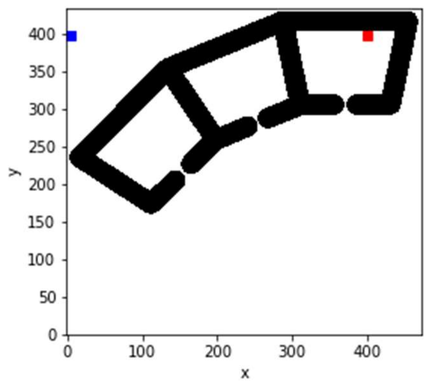
Fail! It can't catch target in 20000 iterations.

MAP 3b



$n = 20, number\ of\ steps = 5773, caught = True$

MAP 3c



$n = 20, number\ of\ steps = 1995, caught = True$

Discussion:

Even though our robot can catch target in the end, it's clear that our robot stays at some locations for a while.

This is because it is trying to update heuristic value for surrounding vertices unitl they are more informed about the environment. I believe switch to LRTA* can improve the situation because LRTA* can update heuristic value more informed.

C. Influence of value n

n represents how many nodes we expand at step 2).

Therefore, we can expect larger n will create a better path for us because it will explore more possible path.

Using MAP 2 to test on n = 7,8, 9, 10.

Result:

$n = 7, number\ of\ steps = 33, caught = True$

$n = 8, number\ of\ steps = 31, caught = True$

$n = 9, number\ of\ steps = 28, caught = True$

$n = 10, number\ of\ steps = 42, caught = True$

Discussion:

It is clear that when n get larger the number of steps decrease. However, when n is 10 the number of steps increase. This is because expanding 10 cells makes robot discover another path and make target move differently. Therefore, the consequence becomes different, it makes more move to catch the target.

V. Reference

1. Professor Nikolay Atanasov's slides (lecture5~8)