ECE 276B
Project 2

Motion Planning

Due Date: Feb 20, 2018

Professor: Nikolay Atanasov

Chuqiao Song

A53239614

Introduction

In the shortest path problem, given same environment but with increasing grid resolutions, I implement Dijkstra, and A star with different weights (1,2,3,4,5) to solve this shortest path. Basically, the main difference between Dijkstra and A star algorithm is Dijkstra does not use heuristic map to select next searching node in graph; however, the A star algorithm really use the heuristic map to 'predict' next node, which somehow decrease the computation cost. With increasing the weight, we can say I am more trust on my heuristic map, and it makes the searching region narrower.

In the motion planner part, I implement RTAA star algorithm to let an agent (blue square) intercept a moving target (red square) in different environment, and it is a series A star search. The main idea behind this is that the heuristic map being updated after each local search towards target, which makes the heuristic map more informed

Problem Formulation

1) Shortest Path

Basically, this is a static shortest path problem, which means I have fixed start point and target point, and the agent at start point moving after the shorted path is found. Additionally, we have to implement some data structure to store the path I choose to move on.

Dijkstra

Open list: Construct an open list from only starting point, **add** searching frontiers (new nodes) into it, expanding a node in open list based on some specs, and **remove** it from open list are the main procedures in iteration.

Record g value: The g value representing the smallest cost from starting point to a specific node should be record as next iteration judging condition.

Stop checker: The stop checker should be set to finish the iterations representing the shortest path found.

A star and weighted A star

Open list: Construct open list from only starting point, then add new nodes and remove nodes from it.

Closed list: Insert the removed node into close list for judging condition.

Record g value: It is similar to the procedure in Dijkstra, record the smallest cost g from start point to a specific node.

Heuristic value: reference the heuristic value with recorded g value to check whether this node should be expanded.

Stop checker: The stop checker should be set to finish the iterations representing the shortest path found.

Different weight: Also try different weight in this algorithm, with weight = 1,2,3,4,5, where weight =1 is just regular A star.

2) Motion planning

Limits of time: Given the location moving target, the agent planner has to finish planning with in 2 second.

Update heuristic map: In each iteration with in 2 second, the heuristic map should be updated to become more informed.

Catching moving target: the target is moving in this part; an online method should be developed here to catch that and avoid local minimum

Next moving location: Given the specs that the agent can only move around nearest 8 points, agent have to decide which location to go to catch the target.

Technical Approach

1) Shortest path

Basically, the dictionary in python is implemented to store the path, that is to say the node with minimum g value or g value plus h value is linked to is children nodes until the target nodes is in the children nodes. Therefore, by knowing this relationship, the tracing back function can be implemented to find the shortest path.

Dijkstra

A. Open list

The open list in the Dijkstra, I use set in python to implement it which can make sure one node can only represent once in Open list, other than list in python, it may cause multiple same nodes which may increase the searching time. By using the Open list, I first have to find node with smallest g value and then remove it from Open list. By expand this node, I can find its adjacent nodes so that we can working on these nodes to check whether these nodes can be added in to the open set. Basically, the nodes remove from open list can never been added into the open list again.

B. record g value

The g value corresponding to each node is first initialized with infinite expect the starting node is set to 0. To update g value of corresponding node is checked by the whether the g value is smaller than present g value, and if it is, update it, otherwise keep present one.

C. Stop checker

The stop check is defined by if the target node is removed from the open set, the search terminates.

A star and weighted A star

A. open list, close list, heuristic value

In general, the main procedure is similar to the Dijkstra. I implement the Open set adding nodes for next iteration expanding, but now we remove the node from open set by check it has minimum $g_i + \varepsilon h_i$in one iteration. Also, the closed set is introduced in here, to store the removed nodes. Similarly, the set is the structure for open and close to avoid searching redundancy.

B. record g value and stopper checker

Similarly, the g value is updated if it is smaller than present value, and the stopper is defined as if targets is in the close set, we terminate the searching

C. trying different weight (ε)

Also, A star with different weights is implemented here to check how the searching performance is. I just simply update the ε, if it is regular A star, ε=1, for weighted A star we can choose ε with arbitrary value. Basically, the lager weight is, the more I trust on the heuristic. It can significant increasing the searching speed by not considering all the nodes in open set, but there is a tradeoff here, it may not find the shortest path.

Following is the algorithm for Dijkstra and A star with or without weight, and corresponding searching space.

Dijkstra

Algorithm 1 Label Correcting Algorithm

1: OPEN $\leftarrow \{s\}$, $g_s = 0$, $g_i = \infty$ for all $i \in \mathcal{V} \setminus \{s\}$
2: **while** OPEN is not empty **do**
3: Remove i from OPEN
4: **for** $j \in$ Children(i) **do**
5: **if** $(g_i + c_{ij}) < g_j$ **and** $(g_i + c_{ij}) < g_\tau$ **then**
6: $g_j \leftarrow (g_i + c_{ij})$
7: Parent$(j) \leftarrow i$
8: **if** $j \neq \tau$ **then**
9: OPEN $\leftarrow$ OPEN $\cup\{j\}$

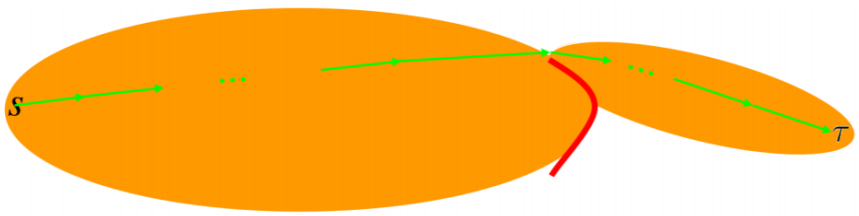
The graph shown above is the searching space for Dijkstra with judging condition g_i , it really like a circle as Dijkstra algorithm expands the pace is not goal-direction orientated.

A star and weighted a star

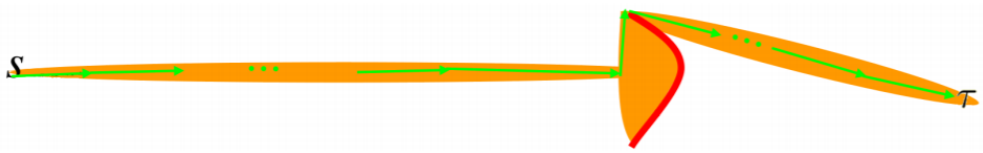
Algorithm 2 Weighted A* Algorithm

```
1:  OPEN ← {s}, CLOSED ← {}, ε ≥ 1
2:  g_s = 0, g_i = ∞ for all i ∈ V \ {s}
3:  while τ ∉ CLOSED do                          ▷ τ not expanded yet
4:      Remove i with smallest f_i := g_i + εh_i from OPEN   ▷ means g_i + εh_i < g_τ
5:      Insert i into CLOSED
6:      for j ∈ Children(i) and j ∉ CLOSED do
7:          if g_j > (g_i + c_ij) then
8:              g_j ← (g_i + c_ij)
9:              Parent(j) ← i
10:             Insert j into OPEN
```

expand state i:
○ try to decrease g_j using path from s to i



The graph shown above is for A star with small weight or weight= 1, and the searching space is expanded from start point to goal with more specific orientation. The function $f_i = g_i + \varepsilon h_i$ is used here to judge which node in the open set to be expand.



The graph shown above is for A star with large weight, as we can see the searching space is much narrower than before, and it really decreases the searching time. To some extend we more trust on the heuristic value h and make the searching more directed.

2) Motion planning

To solve the Limits of time, updating heuristic value map and deciding which adjacent points to go, I implement the Real Time Adaptive A star (RTAA)in this motion planning problem.

Limits of time

RTAA can do the local search instead of doing the whole map search to find the target. That is to say, it first searches small region of map which is decided by the number of look-ahead, which means in normal A star the number of look ahead is infinite but in RTAA the number of look ahead is limit to a specific number. Therefore, in large map, the agent does have to wait until the shortest path is found and in general it may take much long time which means the agent cannot do anything in this time interval. However, RTAA really fixed this problem, it is an online method, and it somehow splits the working load not working for whole map. In this case the agents can decide where to move in within 2 second.

Update heuristic map, Catching moving target:

The main difficulty in this motion planning is that we do not heuristic map and it is always changing as the target is moving. Also, the g value is not given, and we have to build it to find a path from present agent point to some middle optimal point between the agent and the target. The RTAA is really helpful in this condition, it can update the heuristic map and g value map in each epoch (one epoch = one RTAA searching) so that find a path from present agents point to that middle optimal point. That optimal point is the node open list with smallest $f_j = g_j + \varepsilon h_j$ at the end of each epoch. To update the h value of the nodes in open list in each epoch, we apply $h_i = (smallers\ f_j) \quad - \quad g_i$.

Next moving location:

As the agent can move only one step in 2 second, we can choose the position in that path and also in the 8 adjacent points of the agent point as next iteration position. (one iteration = one look ahead within epoch). Meanwhile, the heuristic map is updated and store for next iteration or next epoch using, ideally the more accuracy the heuristic map is, the easier agent can catch the target; Therefore, we have to always update the heuristic map. Generally, heuristic map is only initialized once and being update in each epoch, but g value map is always initialized at beginning of each epoch as and updated in each iteration, as we can always treat the present agent location as a start point to find the shortest path to the target.

Local minimum:

Sometimes the agent will enter into the local minimum and cannot get rid of it, and in this case, we can increase the number of look ahead in each epoch, but there is a tradeoff: it really increasing the computational cost. 1000 look-ahead in one epoch is much slower than 100 look-ahead in one epoch to let agent decide where to move.

In general, the main procedure is similar to the weighted a star, instead, we can update the heuristic map in real time, and split the computation cost for whole map into small region so that let agent act quickly.

Following is the procedure to realize the RTAA star.

constants and functions

S	set of states of the search task, a set of states
$GOAL$	set of goal states, a set of states
$A(\,)$	sets of actions, a set of actions for every state
$succ(\,)$	successor function, a state for every state-action pair

variables

$lookahead$	number of states to expand at most, an integer larger than zero
$movements$	number of actions to execute at most, an integer larger than zero
s_{curr}	current state of the agent, a state [USER]
c	current action costs, a float for every state-action pair [USER]
h	current (consistent) heuristics, a float for every state [USER]
g	g-values, a float for every state [A*]
$CLOSED$	closed list of A* (= all expanded states), a set of states [A*]
$\bar{s}$	state that A* was about to expand when it terminated, a state [A*]

```
procedure realtime_adaptive_astar():
{01} while (s_curr ∉ GOAL) do
{02}   lookahead := any desired integer greater than zero;
{03}   astar();
{04}   if s̄ = FAILURE then
{05}     return FAILURE;
{06}   for all s ∈ CLOSED do
{07}     h[s] := g[s̄] + h[s̄] − g[s];
{08}   movements := any desired integer greater than zero;
{09}   while (s_curr ≠ s̄ AND movements > 0) do
{10}     a := the action in A(s_curr) on the cost-minimal trajectory from s_curr to s̄;
{11}     s_curr := succ(s_curr, a);
{12}     movements := movements − 1;
{13}     for any desired number of times (including zero) do
{14}       increase any desired c[s, a] where s ∈ S and a ∈ A(s);
{15}     if any increased c[s, a] is on the cost-minimal trajectory from s_curr to s̄ then
{16}       break;
{17} return SUCCESS;
```

Results and Discussion

1) Shortest path

Dijkstra, A* and weighted A* algorithm comparison:

Here are two tables illustrating the performance of Dijkstra, A* and weighted A* algorithm by its costs and number of iterations.

	Dijkstra	A*	Weighted A* with $\varepsilon = 2$	Weighted A* with $\varepsilon = 3$	Weighted A* with $\varepsilon = 4$	Weighted A* with $\varepsilon = 5$
Input1	30.14	30.14	30.63	31.80	31.80	31.80

Input2	28.73	28.73	29.80	32.39	32.385	32.38
Input3	26.73	26.73	28.80	31.80	32.09	35.68

Table. 1 Costs of Dijkstra, A* and weighted A* algorithm

	Dijkstra	A*	Weighted A* with $\varepsilon = 2$	Weighted A* with $\varepsilon = 3$	Weighted A* with $\varepsilon = 4$	Weighted A* with $\varepsilon = 5$
Input1	99	63	31	28	29	28
Input2	379	212	102	93	86	79
Input3	1560	693	376	335	293	269

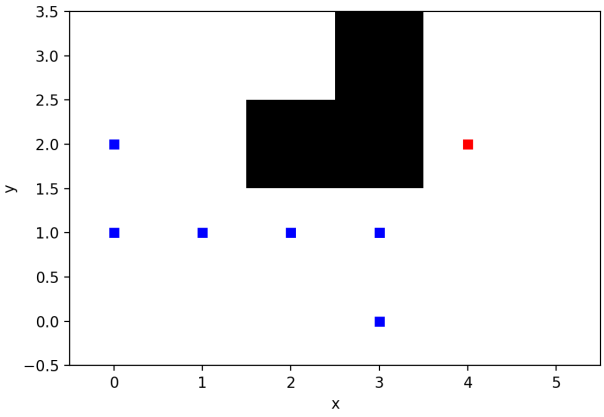
Table. 2 Iterations of Dijkstra, A* and weighted A* algorithm

In general, Dijkstra can produce lowest cost for finding shortest path, but it takes more iterations than A star and weighted A star algorithm. However, by using normal A star, the number of finding shortest reaching to target decrease significantly and with similar shortest path costs. As shown in table 1 Dijkstra cost is same as the A star, but the iterations of A star in table really decreases much. For input 3 cost are both 26.73 but with A star iterations becomes 693 other than Dijkstra 1560. By introducing the weighted A star, with higher weight the iterations decrease further, however, the cost of the found shortest path is increased. As shown in table1 and table2 for input 2, the cost of A star and A star with weight =5 is 28.73 and 32.38 respectively, and number of iterations is 212 and 79.

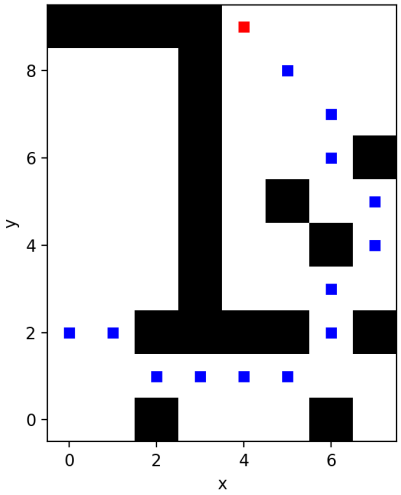
2) Motion Planning

RTAA star algorithm lets us update the heuristic online for moving target such that we can intercept the moving target, and also it splits the whole computation into small parts which can let agent react fast and it is a dynamic algorithm. However, the total computation cost is much larger than regular A star, since there is much redundant computation in RTAA, and sometimes the agent will enter into local minimum as the look ahead region is too small. For example, if the total computational cost for iteration for A star is O(n), then for RTAA is about O (1000 n) as I implement 1000 look ahead in this project. However, the time need to do reaction for RTAA is O(n/1000) comparing with regular A star algorithm O(n).

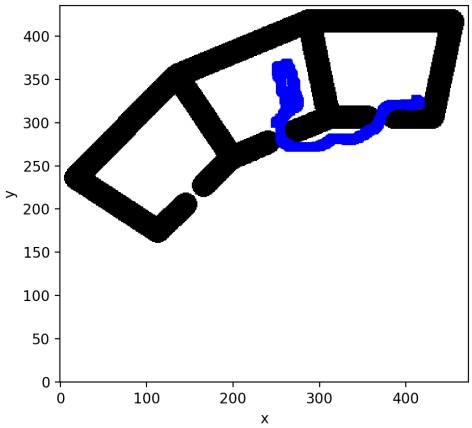
In this project, I implement 1000 look ahead, and it performs well in all the map except in map 1 and map 7, since it is too much large, and it really takes me too much time to catch, and sometimes cannot be catch within specific epochs. The main idea for RTAA is to use informed heuristic map to do moving decision.



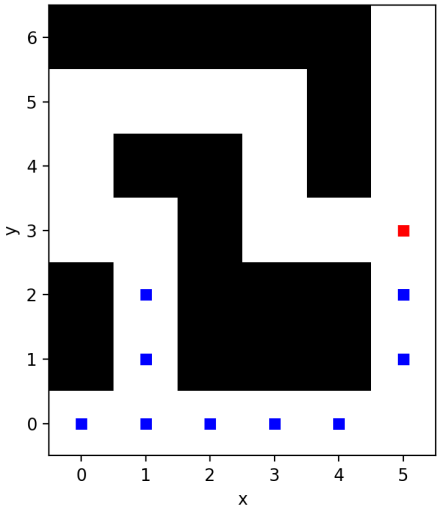
map0



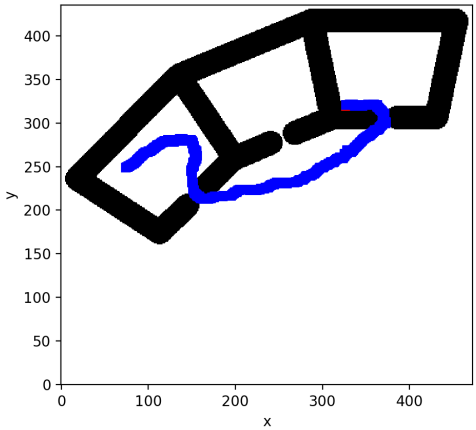
map2



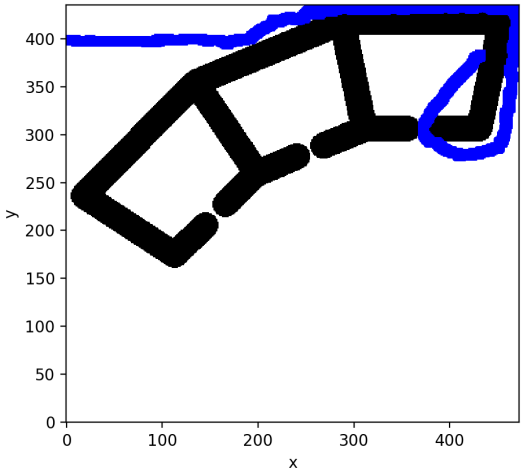
map3



map4



Map3b



map3c

	Map0	Map2	Map3	Map4	Map3b	Map 3c
robot pose	(0, 2)	(0,2)	(249,300)	(0,0)	(74,249)	(4, 399)
Target pose	(5, 3)	(7,9)	(399,399)	(5,6)	(399,399)	(399, 399)
Number iters	5	12	423	13	452	933

The map5, 6 are similar as map 3 and map4, so I have not list statistics here, and map1 and map 7 it takes me too much time, it may catch within specific time, but I have not test it yet.