Catching a Moving Target

Name: Tanmay Chinchanikar

1 Algorithm Details:

1.1 Summary:

The basis for the algorithm implemented is the conventional A* algorithm. To put it in simple words, The algorithm first finds all the possible paths and then picks the optimal solution out of all the available paths. The algorithm pre-computes the path, that is all the planning is done before the robot actually starts moving. The steps/flow of the solution are as follows:
R - Robot
T -Target

- 1. Instead of finding the optimal path directly, A* is used to find the policy that computes the least-cost path from the R's initial position to all the positions available (below threshold) on the map. Basically, all the states in the OPEN list are expanded until none are left.
- 2. Each point on the trajectory of the target is assumed to be a possible position of intersection for R and T. Therefore, for every position on the T's trajectory, the least cost path from R is calculated by backtracking. Instead of saving the path, only the number of steps required (Time of traveling) for R (R_{steps}) is saved.
- 3. For each position mentioned above (each location of target trajectory), the number of steps (Time for traveling) required for the T (T_{steps}) to reach that exact position is calculated. If (T_{steps}) $\geq$ (R_{steps}), it is in fact possible for the R and T to meet at that location.
- 4. The total cost for such points is calculated using the following formula:
  Cost = g-cost + Time$_{diff}$*Cost$_{cell}$

  Where:
  Cost = Total cost to be minimized for better solution
  g-cost = g-cost for that position (Minimum cost required to reach there from R's initial position)
  Time$_{diff}$ = (T_{steps}) $\geq$ (R_{steps}) (Waiting time for R)
  Cost$_{cell}$ = cell/edge cost of that particular location.
- 5. Out of all the filtered positions from step 3, the position with the total minimum cost (calculated in step 4) is chosen.
- 6. The path is backtracked for this location and this time, the path is saved and provided to R.
- 7. Once R reaches that position, it is then instructed to wait there until T arrives.
- 8. In the case that not even a single position on the target trajectory is found in step 3 that satisfies the condition, Instead of normal A*, the algorithm is shifted to weighted A* with weight =10 and the whole process is recursively repeated with increasing weight factor until at least one solution is found.

1.2 Classes/Data Structures:

- state class: A custom class that represents properties of a free/non-obstacle cell in the map.
  class Variables:
  - robotposx: x coordinate of cell
  - robotposy: y coordinate of the cell
  - g-cost: g cost of the cell as part of path planning
  - h-cost: heuristic cost of the cell as part of path planning
  - f-cost: g-cost + h-cost
  - state-expanded: If the cell has been expanded
  - parent: The parent of the current cell as part of path planning
- unordered-map: Map that saves the state information for every x,y coordinate if it has been visited once.
- Priority-queue: To select/choose the state to be expanded (State with minimum cost)
- Stack: To backtrack and use a particular path.
- chrono: To calculate the time required for the processing algorithm.

1.3 Other Information:

- The heuristic function used for normal A*:

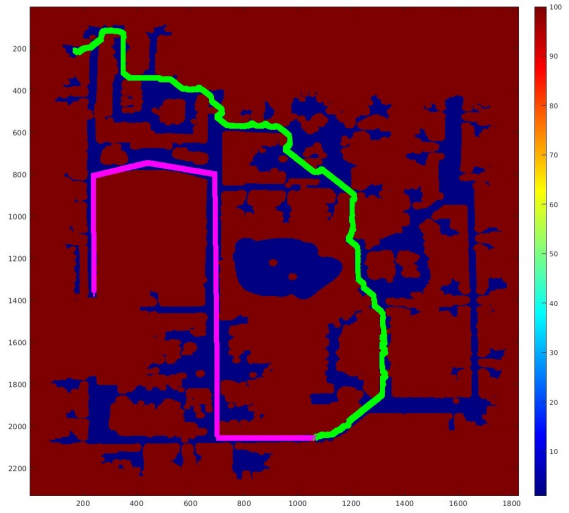
$$H = \sqrt{2} * MIN(abs(dx), abs(dy)) + ((MAX(abs(dx), abs(dy))) - (MIN(abs(dx), abs(dy)))) \quad (1)$$

  The heuristic function used for weighted A* is the diagonal distance heuristic.
- The algorithm considers the processing time that is already used before performing any time-dependent process in the algorithm. This helps in handling the edge cases (where even 1 sec could make a difference). (I put a piazza private question regarding the same (https://piazza.com/class/l7a2ri0d4u8mg/post/41)).
- All the class objects are dynamically allocated in the algorithm. As soon as that particular object is useless, it is deleted before starting the next process. All the stacks/vectors/queues are also emptied whenever their work is done.
- The algorithm is a very generalized algorithm. It can be used to find a path to the static as well as a dynamic target at any position on the same map. It switches to weighted A* in case normal A* doesn't work and keeps increasing the weight until a path is found. It can also save the number of steps required for the robot to reach the target. In the worst case where even increasing weight doesn't work, beyond a particular weight limit (Kept 90 here randomly), the robot goes straight to the last position of the target greedily while avoiding the obstacles.
- Compiling the code:
  The default compiling command: ***mex planner.cpp*** should compile the code without any error.

Note: I have added some last-minute modification code to incorporate the worst/edge cases and make the algorithm more generalized. Please let me know if for some unknown reason there are any compilation/planning/random issues/bugs while testing. I will share access to all the previous stable GitHub commits done before the deadline.
Here's the link for my private GitHub repository for homework:
`https://github.com/tan-may16/16782_HW1/tree/solution`

2 Reference Map results:

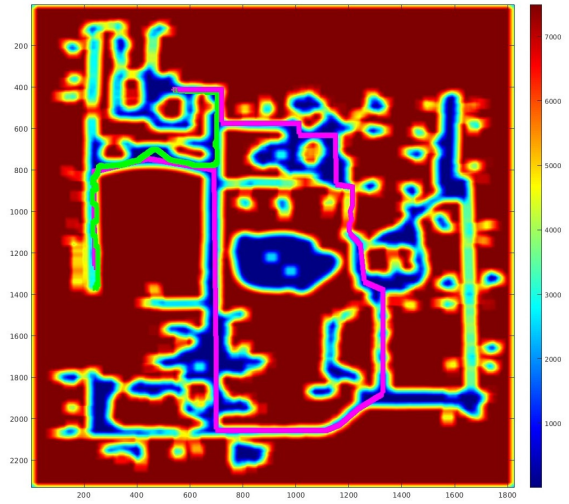
The robot successfully caught the target on all the reference maps.
Visualizations and path cost results of the reference maps are on the next page.



(a) Target-Robot intersection - Map1

```
>> runtest('map1.txt')
Intersection at target_traj index =2641 ,Cost Estimate:2640

RESULT:
        target caught = 1
        time taken (s) = 2641
        moves made = 2639
        path cost = 2641

ans =

  logical

   1
```

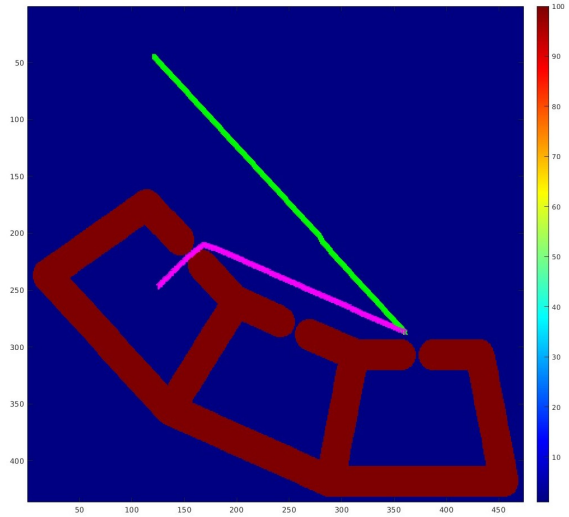
(b) Path Cost - Map1



(c) Target-Robot intersection - Map2

```
>> runtest('map2.txt')
Intersection at target_traj index =5013 ,Cost Estimate:1990516

RESULT:
        target caught = 1
        time taken (s) = 5013
        moves made = 1529
        path cost = 2000630

ans =

  logical

   1
```

(d) Path Cost- Map2

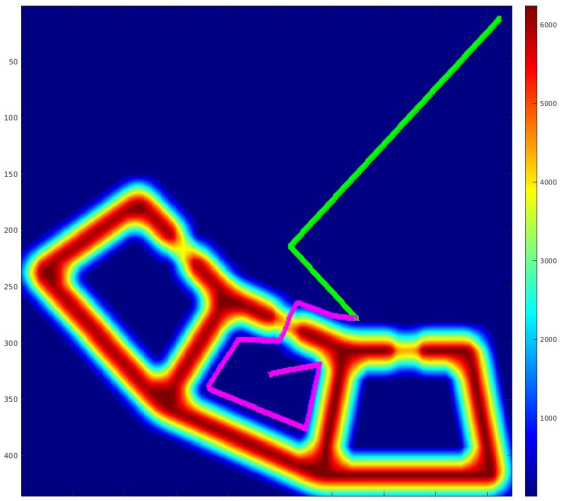


(e) Target-Robot intersection - Map3

```
>> runtest('map3.txt')
Intersection at target_traj index =244 ,Cost Estimate:243

RESULT:
        target caught = 1
        time taken (s) = 244
        moves made = 242
        path cost = 244

ans =

  logical

   1
```

(f) Path Cost - Map3

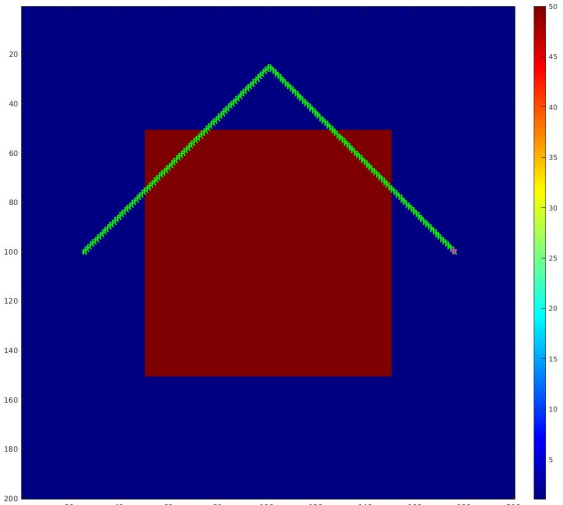
Figure 1: (a),(c),(e): Visualization – Robot catching moving Target for maps 1, 2 and 3
(b),(d), (f): Path costs – Robot catching moving Target for maps 1, 2 and 3



(a) Target-Robot intersection - Map4

```
>> runtest('map4.txt')
Intersection at target_traj index =380 ,Cost Estimate:379

RESULT:
        target caught = 1
        time taken (s) = 380
        moves made = 266
        path cost = 380

ans =

  logical

   1
```

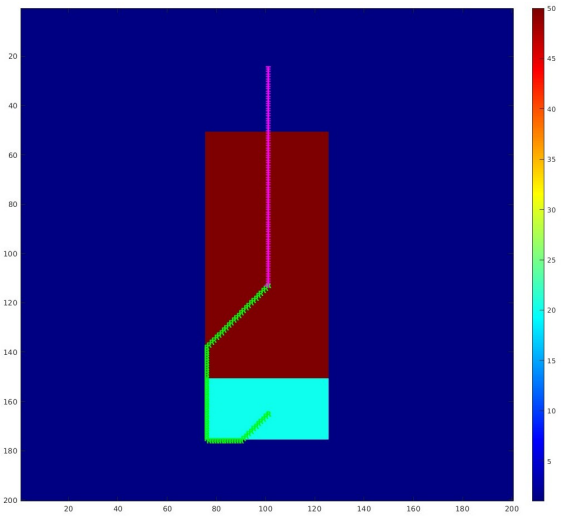
(b) Path Cost - Map4



(c) Target-Robot intersection - Map5

```
>> runtest('map5.txt')
Weight = 10
Weight = 20
Weight = 30
Weight = 40
Weight = 50
Intersection at target_traj index =152 ,Cost Estimate:2552

RESULT:
        target caught = 1
        time taken (s) = 151
        moves made = 150
        path cost = 2552

ans =

  logical

   1
```

(d) Path Cost- Map5



(e) Target-Robot intersection - Map6

```
>> runtest('map6.txt')
Intersection at target_traj index =89 ,Cost Estimate:1552

RESULT:
        target caught = 1
        time taken (s) = 89
        moves made = 87
        path cost = 1542

ans =

  logical

   1
```

(f) Path Cost - Map6

Figure 2: (a),(c),(e): Visualization – Robot catching moving Target for maps 4,5 and 6
(b),(d), (f): Path costs – Robot catching moving Target for maps 4,5 and 6