16-782: Planning and Decision-making in Robotics

Name: Kshitij Kabeer, Andrew ID: kkabeer

Approach - Iterative A* with varying transition costs and goals

Step 1: Finding an Appropriate Goal Location

First, I decide the goal location that the robot should try to reach. The algorithm checks for all the locations of the target, starting backwards. It uses the Chebyshev distance to decide whether it is possible to reach a certain goal location. Basically, if the Chebyshev distance is greater than the maximum time available to the robot to reach that goal location, there is no way that the robot can possibly reach that goal location.

Keep Re-Planning, trying out different goal locations.

Once a possibly feasible goal location is found, the planner plans a path to the goal location. However, it doesn't do it just once at the beginning, it keeps re-planning to find a better goal location. The planner is written in such a way that it can pause and resume the planning process. The planner always pauses before a time limit of 750 milliseconds, and continues in the next iteration. Hence, once a feasible plan is found, the robot can keep following the original plan which is fed to action_ptr after the planner pauses, without waiting for planning to finish.

The planner keeps re-planning the path to the various goal locations in descending order of their priority. A goal location P is higher priority than another goal location, if the target reaches P after it reaches Q, since it means that the robot would most likely be able to catch it. In each re-plan, the planner plans a path from the estimated robot position to the goal location. It uses the time taken for planning in the previous iterations as an estimate of when the planning will finish, and uses that to decide what the estimated robot position is. So if it took 7 seconds to plan the path previously, the planner will plan a path from the robot's expected position 7 seconds later to the goal location. The expected position of the robot 7 seconds later is calculated from the previous plan.

If it finds a lower cost path, it replaces the corresponding portion of the previous plan with the new path. It has some checks to ensure that the path is never replaced if the robot has already moved past the estimated robot start position, which might happen if the planning took longer than expected.

Use A* with increasing transition cost

In each iteration of planning, a 2D A* search is conducted at most 5 times. Each time, the edge cost of making each transition is increased by 25% of the collision threshold. The reason I am doing this is because sometimes the solution found by A* has way too many steps, and by the time the robot executes those steps, the target is already gone. By increasing the penalty on transition brought about by increasing the edge cost, the search begins to prefer a lower number of transitions (i.e. steps) to reach the goal. I start with a penalty of 0, and I keep increasing the penalty in increments of 25% of the collision threshold until I have a maximum penalty on state transitions to be equal to the collision threshold. I keep doing this until the planner finds a feasible solution to the goal location.

Heuristic Function, Transition Cost And Guaranteed

The heuristic function I am using is the Chebysev distance, with both the diagonal as well as side transitions cost the same. This is an consistent and admissible heuristic function, as the distance to a goal is upper bounded by the Chebyshev distance.

The transition cost I use c(s, s') = CELL_COST[s'] + penalty. This is not the actual cost to take the transition, but as mentioned earlier, the penalty is added to bias the search in favor of shorter paths. The Chebyshev distance is still admissible and consistent in this case.

Hence, the paths from point A to point B are guaranteed to be optimal, but the overall solution is not guaranteed to be optimal. This is because the goal location that the robot chooses to go to might not be the best goal location to go to.

One guarantee is that replans always result in the path becoming better or staying the same. In no case will a path replacement occur that would cause the total cost to go up. In other words, the changed path will always be lower cost or the same cost as the path generated by the planner in a previous iteration.

How are the nodes stored

I have used an Adjacency list (called node_grid_) to store the nodes on the graph. Each node consists of a lot of information, like location(x and y), a pointer to its parent, f,g,h values, among other things. I have also used C++'s memory containers, namely std::shared_pointer to dynamically allocate Nodes, and manage pointers to different Nodes. This is basically for preventing memory disasters like memory leaks or dangling pointers. Even though the node_grid_ looks 3D, it is actually just one 2D layer. I initially wanted to plan in 3 dimensions (x,y,time) but that didn't work out because of memory and time constraints. So, I just have one 2D layer and I just use t=0 to reference that layer.

Run Instructions

The running instructions are the same as the instructions given with this assignment.

Results

The following section details the results. I have included the output of the planner. I have also included the images of the robot and target trajectories, both separately and also on the same map because in some of the maps one trajectory overlaps with another.

Map1

Output is

```
target caught = 1
time taken (s) = 5344
moves made = 421
path cost = 5344
```

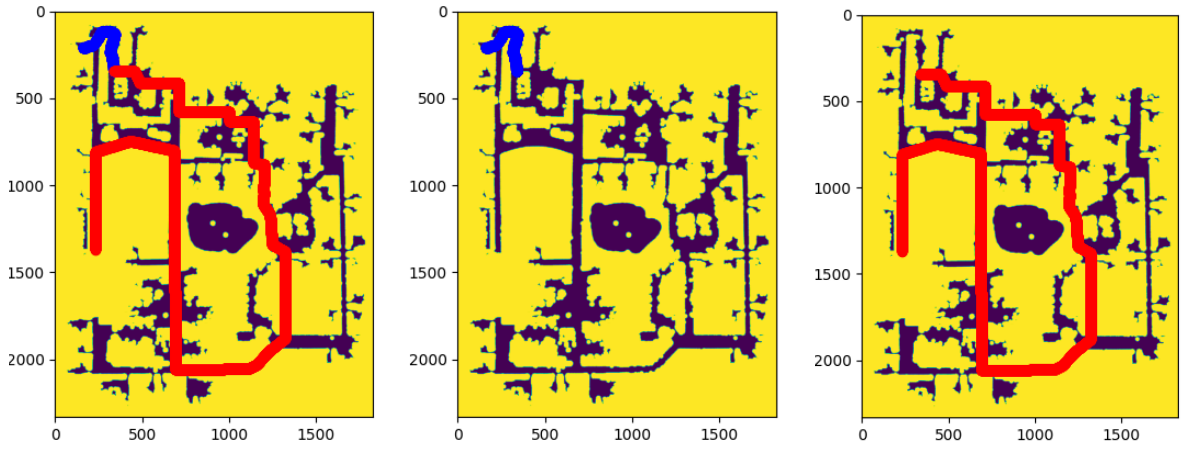

Figure 1: Both trajectories, Robot trajectory, Target trajectory

Map2

Output is

```
target caught = 1
time taken (s) = 5012
moves made = 1528
path cost = 2000630
```

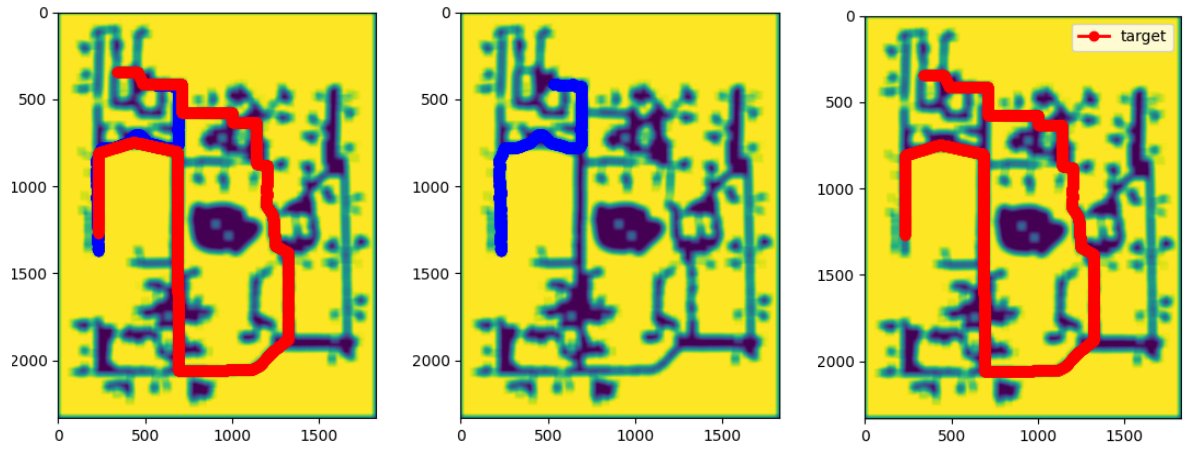

Figure 2: Both trajectories, Robot trajectory, Target trajectory

Map3

Output is

```
target caught = 1
time taken (s) = 791
moves made = 283
path cost = 791
```

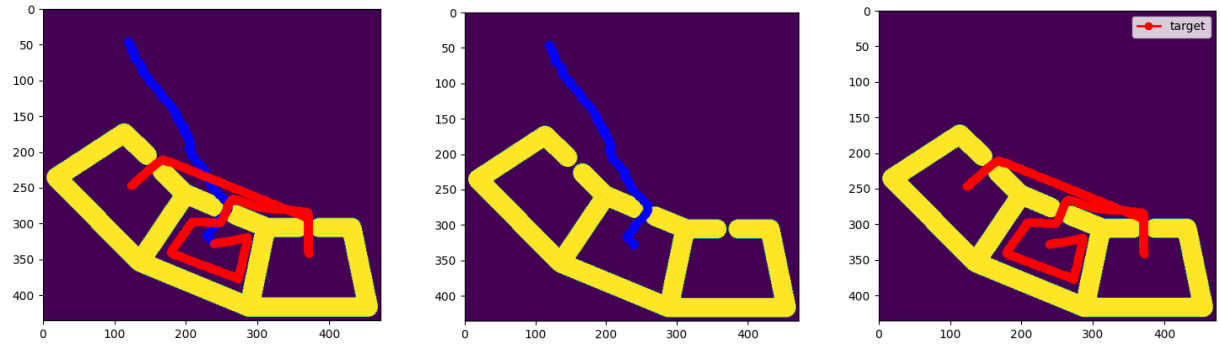

Figure 3: Both trajectories, Robot trajectory, Target trajectory

Map4

Output is

```
target caught = 1
time taken (s) = 696
moves made = 254
path cost = 696
```

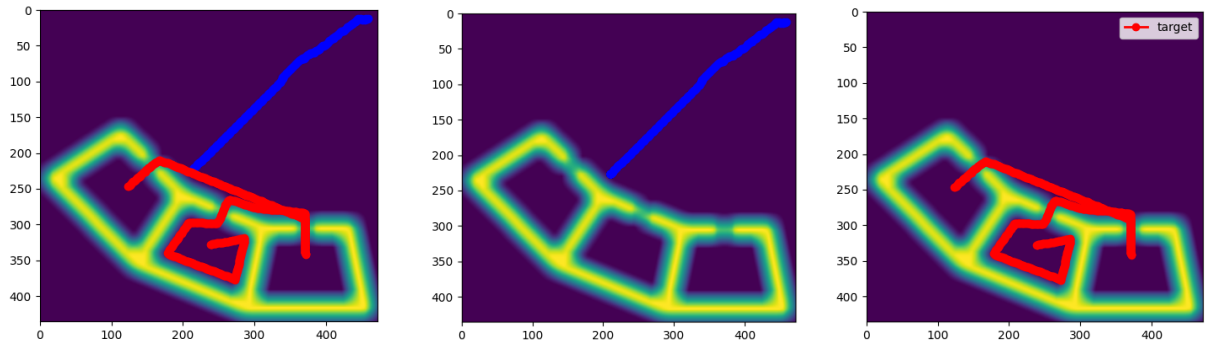

Figure 4: Both trajectories, Robot trajectory, Target trajectory

Map5

Output is

```
target caught = 1
time taken (s) = 150
moves made = 150
path cost = 2551
```

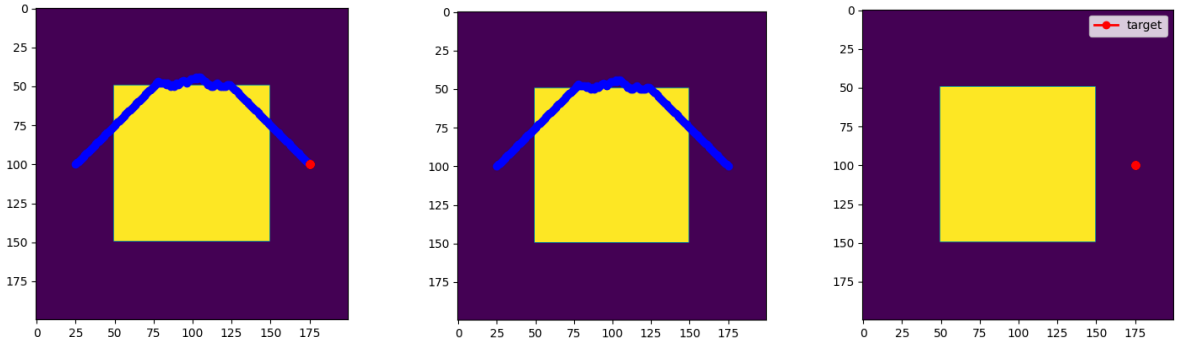

Figure 5: Both trajectories, Robot trajectory, Target trajectory

Map6

Output is

```
target caught = 1
time taken (s) = 140
moves made = 0
path cost = 2800
```

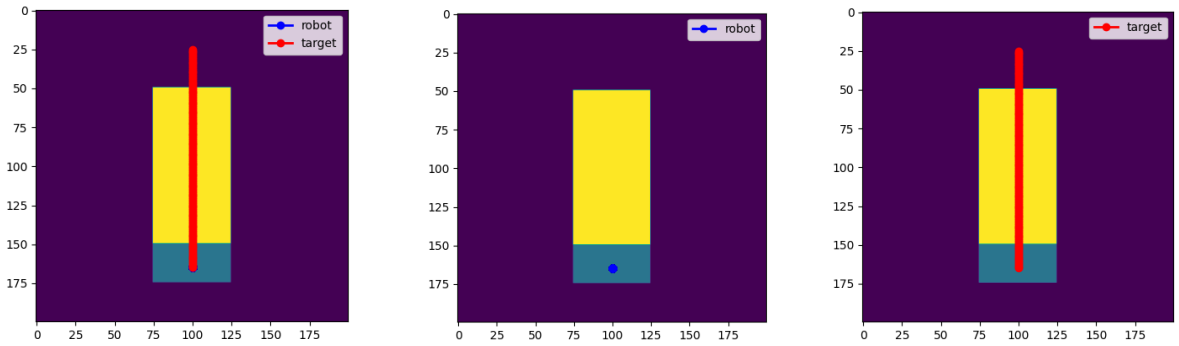

Figure 6: Both trajectories, Robot trajectory, Target trajectory

Map7

Output is

```
target caught = 1
time taken (s) = 300
moves made = 300
path cost = 300
```

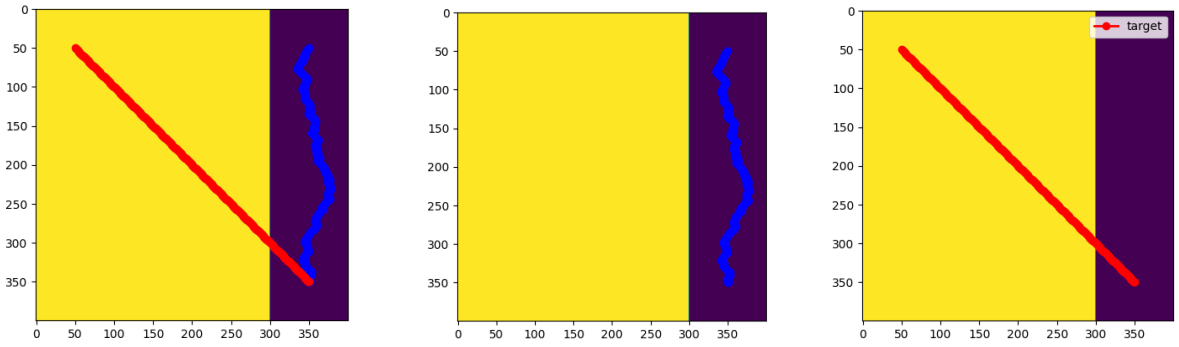

Figure 7: Both trajectories, Robot trajectory, Target trajectory

Map8

Output is

```
target caught = 1
time taken (s) = 451
moves made = 410
path cost = 451
```

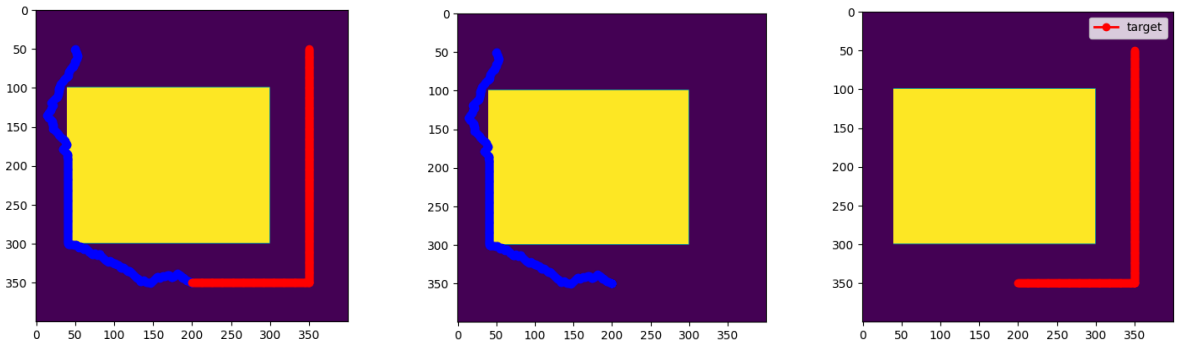

Figure 8: Both trajectories, Robot trajectory, Target trajectory

Map9

Output is

```
target caught = 1
time taken (s) = 600
moves made = 350
path cost = 600
```

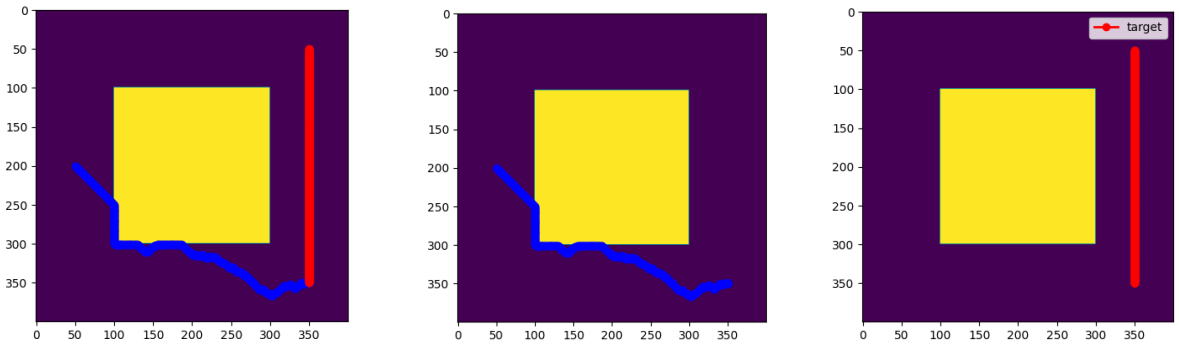

Figure 9: Both trajectories, Robot trajectory, Target trajectory

Appendix

Even though the approach detailed above is my final submission, I did try lots of different things that didn't work out in the end. I wanted to write about those approaches as well, since I put a lot of effort into it. The code has been included in the submission as well, with each cpp file appended with a version number. V1 is just a previous version of the approach described above. V2 and V3 correspond to the two other approaches I tried, which are detailed below. Feel free to skip this section if you want to.

Multi-Goal 3D A* planning

I tried planning in 3 dimensions - x, y and t, which I also made multi-goal. Each point of the target trajectory, along with its time step, became goals for the planning problem. However, as I quickly found out, this problem is computationally extremely heavy. I tried to come up with intelligent heuristics to guide the search, but it was still either taking way too much time or was just missing the goal point entirely. I also tried different weights to the heuristics. Some of the heuristic functions I used:

- Using the manhattan or chebyshev distance to the goal point closest to cell. This ignores goal cells that are too far to reach. Returns infinity(DBL_MAX) if there is no goal cell that can be reached from this node.
- Using the average manhattan or chebyshev distance to all the goal points. Once again, goal cells too far to reach are ignored.
- A binary heuristic which returns 0 if the robot can possibly reach any of the goal cells, infinity otherwise (This should theoretically be admissible and consistent)
- And some other really bad ones, like the number of reachable goal points from the current cell. This was a very bad heuristic, since it didn't like nodes that are really close to a few goal points but very far from or unreachable to others.

This approach is able to catch the target for map 3, 5 and 6. For the rest of the maps, either it takes a long time to finish executing (which means it would have already missed the target) or doesn't catch it. The planner also performs better than the submitted planner on map 3, achieving a cost of 391. The code is in planner_v2.cpp and planner_v2.hpp. To build, use the command:

```
g++ runtest.cpp planner_v2.cpp
```

To run, the instructions are the same as the instructions provided with the assignment.

Learning Real Time/Real Time Adaptive 3D A*

This has not been taught yet, but I did find Prof. Likhachev's lectures from previous classes online. These approaches seemed perfect for the problem at hand, allowing me to utilize all the time available to keep making the plan better, while also simultaneously moving the robot. However, I didn't anticipate so many pitfalls in the approach that would make it unsuitable for the problem at hand, and the implementation also became quite messy and difficult to test. I used all the possible heuristic functions mentioned in the previous approach.

Firstly I tried implementing Learning Real Time A* for this problem. Instead of fixing the number of expanded states in each iteration, I just fixed a time bound for the expansion of states. I then implemented backtracking to correct all the h-values. I didn't really quite understand the answer to the question "Does it matter in what order?" In Prof. Likhachev's lecture, I am guessing that we backtrack in the reverse order of node expansion, but I am not really sure. I am also not sure whether you correct the h-values of all the states in the closed list (which grows really large really fast), or only the states that were transferred to the closed list in this iteration? I tried both these things, but none worked. Either the robot would get stuck somewhere and not move, or would sometimes not be able to catch the target. Sometimes the robot would even move in a direction that took it away from all of the target locations, which baffled me.

I also tried Real Time Adaptive 3D A* since the h values were easy to calculate and I didn't really need to bother with the order in which I correct the g values. It didn't work either. The same issues that plagued LRTA* also plagued this one.

After all this testing I realized that I had missed a couple of key points because of which this approach might not work here. Some of the conditions mentioned in the lecture were

- **All robot moves are reversible** - This is not true for this problem, since the node now also consists of a time element, and it is not possible for the robot to go backwards in time. In fact, all the transitions are unidirectional since time always moves forward.
- **All costs are bounded from below with $\Delta > 0$** - This is not true as well, since many cells have a traversal cost of 0. There could be scenarios where the cost to reach a particular cell is entirely 0. I tried circumventing this by adding a very small non-zero cost to each transition, so that the cost of traversing a path is always greater than 0. But it didn't work.

I couldn't really figure out how to mold the planning problem in a way that it could fit these constraints. I also couldn't really figure out how to move the robot - Does the robot need to move along the entire planned trajectory before beginning another round of node expansions and h updates, or is it okay to move it one cell and then begin expanding again? I guess a lot of these questions will be answered in the upcoming classes.

The code works on map 5, and achieves a better plan than both the approaches above. But that is it, it either runs for an incredibly long time, or doesn't catch the target in others.
The code is in planner_v3.cpp and planner_v3.hpp. To build use the command:

```
g++ runtest.cpp planner_v3.cpp
```

To run, the instructions are the same as the instructions provided with the assignment.