Summary of Approach

A* Algorithm:

I implemented an A* algorithm incorporating a multi-goal approach in 2D, taking into account all the potential goals along the target's trajectory. My A* algorithm was implemented in two dimensions, with the x position and y position defining the current state.

For the multi-goal approach I considered all points along the target's trajectory as potential goals. When the planner finds a point along the target's trajectory, it checks if that point can be moved to or not. I'm keeping track of the overall number of seconds passed (wall clock time). I also wrote a function to calculate the number of steps the planner determined it would take to plan to that point. Because we also know how long it takes the target to reach that point (based on the number of steps it has taken), I am then able to figure out whether or not the robot will reach that goal in time. If the point can be reached by the robot, then the planner will set that point as the chosen goal, and move on to backtracking the search and sending the robot there. If the point cannot be reached in time, then the planner will continue searching for another goal.

Although I experimented with the 3D implementation of this approach, I found a large trade-off in time. My 3D implementation would not scale for larger maps (1 and 2) but was able to solve maps that needed to enter high cost regions to quickly catch the target in a short amount of time (maps 5 and 8). Therefore, I chose to continue with my 2D weighted A* multi-goal implementation because it could better scale for large maps.

To handle the maps where the 2D A* approach was not catching the target (maps 5 and 8), I implemented a backup greedy A* method. This approach just uses the euclidean distance as the cost rather than taking into account the map cost. This approach provides an optimal path in terms of time but not in terms of cost. If there is a case in which the target cannot be caught in time when using the A* approach which is minimizing cost, then it can default to this method to at least catch the target.

Heuristics:

I experimented with multiple possible heuristics as I was developing my algorithm. Below is a list of the heuristics I tested out. I decided to use an average weighted distance to all the goal states (option 5 below) since that option yielded the best results.

1. Euclidean distance to last goal: For this heuristic option, I just took the euclidean distance from the current state to the final state of the target (last point along the target's trajectory).
2. Weighted euclidean distance to last goal: The option was the same as the one above, but I added in a weight value to scale the heuristic by.
3. Diagonal distance to last goal: The diagonal distance is typically a better measurement of distance than euclidean distance when using an 8-connected

grid. Therefore, I tested out this approach, but the results were very similar to using euclidean distance.

4. Average distance to all goals: After I implemented my multi-goal approach, it no longer made sense to base my heuristic just on the final possible goal. I changed my heuristic to calculate the distance to each point along the target trajectory and take the average of all those distances.
5. Average weighted distance to all goals: The heuristic above did not perform well for map 9, and I realized I needed to more heavily weight the later points along the target's trajectory and weight the earlier points along the trajectory less heavily since the robot likely won't be catching the target there. I'm still calculating the distance to each point along the target's trajectory, but now I'm scaling each distance relative to the timestep that the target will be at that point at. Therefore, the final point on the target trajectory would have the highest weightage, and the first point would have the least weightage. This heuristic worked well for me and helped lower my cost and time values.

Data Structures:

I used a variety of data structures from the standard library to complete this assignment. Below is a list of some of the data structures I used to store relevant information for completing the A* search.

- State: I used a struct to store the state information. The state struct had x and y as attributes.
- Open list: For the open list values, I used a priority queue of types pairs that stored the f-value and the corresponding state. I implemented a custom comparison function to make the top priority values be the lowest f-values. The priority queue made it simple to find the state with the smallest f-values from the open list.
- Closed list: I stored the closed list as an unordered set. I made a custom hash for the state struct which I also used for the unordered maps described below. The unordered set for the closed list was nice to use since I could just check if a state was in the unordered set to determine if it was in the closed list or not.
- G values: My g-values are stored in an unordered map using the custom State struct hash function I made. Each state struct is mapped to a g-value.
- Parent list: My parent list is storing the connections between current states and their parent states. Each successor state struct is mapped to the state it came from. This is stored as an unordered map using the state hash function again. This data structure is used to backtrack the path after the A* search is complete.

Efficiency Tricks:

I employed several efficiency tricks to optimize the A* search described below:

1. Pruning: I avoid expanding states that have already been visited and are in the closed list.
2. Time limit: I have a time limit on my A* search to ensure that the search does not exceed past the amount of time the robot has to move to the trajectory.
3. Early goal detection: I check if a potential goal state can be reached within the available time frame. If it can, I consider it the chosen goal state and move there.
4. Dynamic time: Although the planner algorithm is only using x position and y position in the state, the planner still adapts to the passage of time to ensure that it operates within the specified time limits. I'm keeping track of the number of seconds that have passed and calculating the number of moves it would take my robot to reach a goal as well as the number of moves the target takes to reach that goal. Using this information, I'm always ensuring that the robot can make it to the chosen goal state in time. If the robot cannot make it anywhere in time (such as for maps 5 and 8), it will just use the final position of the target trajectory as the goal position and move there greedily.

Other Approaches Explored:

1. While I was developing this planner, I experimented with many different approaches. I tried scaling up my 2D implementation to a 3D implementation to include time in the robot state. I was able to get this method working, but it solved fewer maps than my 2D approach did due to time and memory constraints. In the 3D approach, I was limited by computation time. To solve this issue, I tried heavily weighting the heuristic in order to expand fewer nodes of the map. However, I was not able to succeed with that method, so I continued with my 2D implementation.
2. My planner utilizes an implicit graph for the search. I chose this method since it scales well for larger maps. I don't need to pre-allocate the entire size of the map beforehand. In terms of memory management, the implicit graph approach is better. However, I'm using the unordered map data type in C++ which uses hash functions for lookups, which can be slow. I thought that this could have been causing my issues with the 3D method not being fast enough. I remade my A* planner using an explicit graph with vectors where I initialized vectors to the size of the map. However, this approach led to running out of memory. Then, I tried storing the memory on the heap instead of on the stack by using the "new" keyword. Unfortunately, I was still having issues with the larger maps, so I went back to my implicit graph approach using hash maps as it was more reliable. If I had more time, I would like to continue exploring this approach since I think it could work well for the 3D implementation.

Map Results

Map1:
target caught = 1
time taken (s) = 5317
moves made = 425
path cost = 5317

Map2:
target caught = 1
time taken (s) = 4672
moves made = 1272
path cost = 6052548

Map3:
target caught = 1
time taken (s) = 462
moves made = 222
path cost = 462

Map4:
target caught = 1
time taken (s) = 637
moves made = 239
path cost = 637

Map5:
target caught = 1
time taken (s) = 150
moves made = 150
path cost = 5050

Map6:
target caught = 1
time taken (s) = 140
moves made = 0
path cost = 2800

Map7:
target caught = 1

time taken (s) = 251
moves made = 251
path cost = 251

Map8:
target caught = 1
time taken (s) = 451
moves made = 410
path cost = 451

Map9:
target caught = 1
time taken (s) = 424
moves made = 350
path cost = 424

Compiling the Code

The code can be compiled and run as described in the homework and below.

To compile the cpp code:
>> g++ runtest.cpp planner.cpp
To run the planner:
>> ./a.out map3.txt;
To visualize the robot and target's trajectory:
>> python visualizer.py map3.txt;