Assignment 1: Robot Chasing Target

Name: Aman Manish Chulawala
Andrew ID: `achulawa`

1 Algorithm

The "robot chasing target" problem involves a scenario where a robot is tasked with tracking and intercepting a moving target. This code implements a path planning algorithm using the A* search method. Here's a breakdown of the key components:

Includes and Definitions: Standard C++ libraries like stdio.h, math.h, and containers like set and vector are included. There are also macros and global variables defined for various purposes.

Node Struct: Defines a struct named Node that represents a node in the A* algorithm. It contains information about the node's position, cost values (g, h, f), and a pointer to its parent node. It also defines a comparison operator for sorting nodes in sets.

Utility Functions:

isValid checks if a given coordinate is within map bounds and is not obstructed.
calculateHeuristic estimates the remaining cost to reach the goal from a given position.

A Search Algorithm*:
A_star is the main A* search function. It takes parameters related to the map, starting and goal positions, and performs the A* search, returning a pair consisting of a vector representing the path and an integer representing the cost of the path.

Planner Function: Planner determines the robot's next action. It takes information about the map, current positions, target trajectory, and other details. It updates action_ptr to provide the next action for the robot. Execution Flow: The planner function checks the PATHFLAG to decide whether to compute a new path or follow an existing one. If a path is being computed, it calls the A_star function and sets action_ptr to the first step of the resulting path. If a path already exists, it continues following it until completion. When the path is complete, it starts planning ahead for the next steps.

In summary, this code implements a path planning algorithm for a robot in a grid-based environment, possibly considering obstacles. It uses A* search with various utility functions and data structures to find an optimal path from a start to a goal position.

The idea behind the flow: Because of the diversity of the maps provided, several iterations of the planner were created. They included several objectives, like minimising cost, path and time. While some planners performed well in some scenarios, they were not all able to deliver results on all maps. As a result, a combinatory solution was created which guaranteed a path, albeit a less optimal one. The allowed the planner to be more generalised. The details of the heuristic and travel costs are discussed below. The way the current planner works is to have the most expensive search at the first step, where it finds the solution to the last step of the target trajectory. This path is executed to completion. Once the robot reaches the end of the target trajectory, it follows the trajectory aggressively upstream, until it is 10 units within the target. At this point, it executes a stop and holds position until intercept. This method guarantees an intercept, albeit at the cost of sub-optimality for smaller maps. The code structure is optimised to an extent that it delivers the fastest performance of the planner. As stated earlier, the most expensive step is the first planning step.

2 Heuristic and Travel Cost

The octile heuristic is a distance estimation method used in grid-based pathfinding algorithms like A*. It is particularly suitable for movements in eight directions (hence the name "octile"). This heuristic is based on the Chebyshev distance, which calculates the maximum of the absolute differences in coordinates. In grid-based environments, the octile heuristic accounts for diagonal movements, making it more accurate than Manhattan distance, which only considers orthogonal movements.

$$h(x, y) = \sqrt{2} \cdot \min(|x - x_{goal}|, |y - y_{goal}|) - \min(|x - x_{goal}|, |y - y_{goal}|) + \max(|x - x_{goal}|, |y - y_{goal}|)$$

Weighted A star with a weight of 3 for the heuristic was used for calculating the total cost.

$$f = g + 3 \cdot h$$

Here, the transition cost g was defined as

$$g_{\text{new}} = g_{\text{parent}} + 1 + \frac{\text{map value}}{\text{collision threshold}}$$

3 Results

The results from the nine test maps are tabulated below.

Map	Time Taken	Moves Made	Path Cost
Map 1	2889	2887	2889
Map 2	3447	3498	9663174
Map 3	558	558	558
Map 4	565	565	195728
Map 5	150	150	5050
Map 6	140	140	2800
Map 7	298	298	0
Map 8	435	432	0
Map 9	502	447	0

Table 1: Performance in the Environments

4 Running Instructions

The zip file has all the work for this assignment. What we are interested in the **planner.cpp**, **runtest.cpp** and v**isualizer.py** scripts. All the other versions of the planner (V1 through V6 can be ignored). In the terminal, navigate to the assignment directory and run the commands to compile and run the maps. The commands I used are:

Compile the code : `cl runtest.cpp planner.cpp`

Run the planner : `./runtest map1.txt`

Run Visualisation : `python .\visualizer.py map1.txt`

The entire assignment was done on a Windows machine using Microsoft Visual Studio Code.

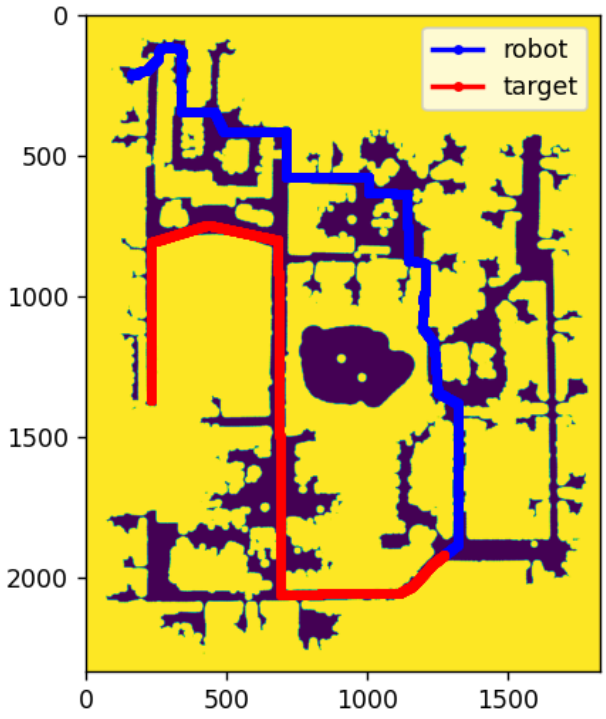

Figure 1: Map 1

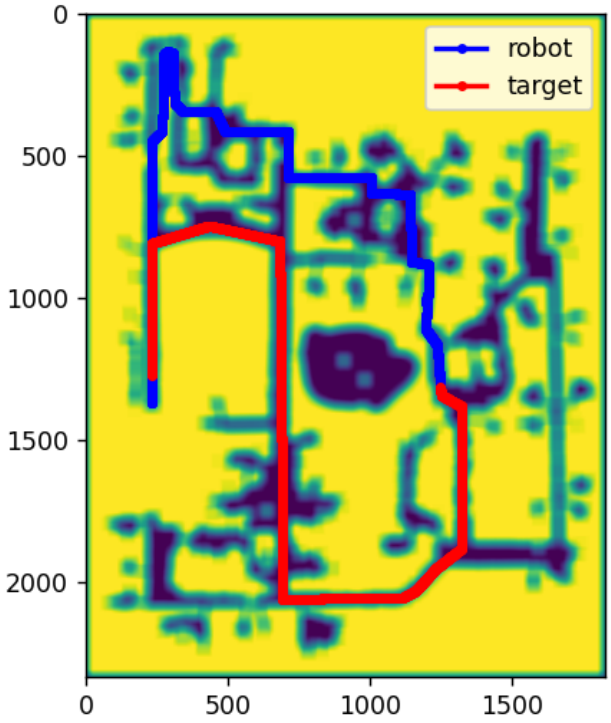

Figure 2: Map 2

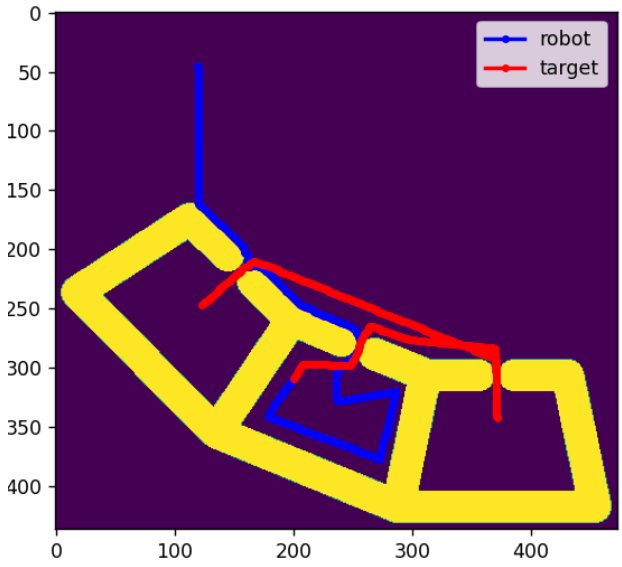

Figure 3: Map 3

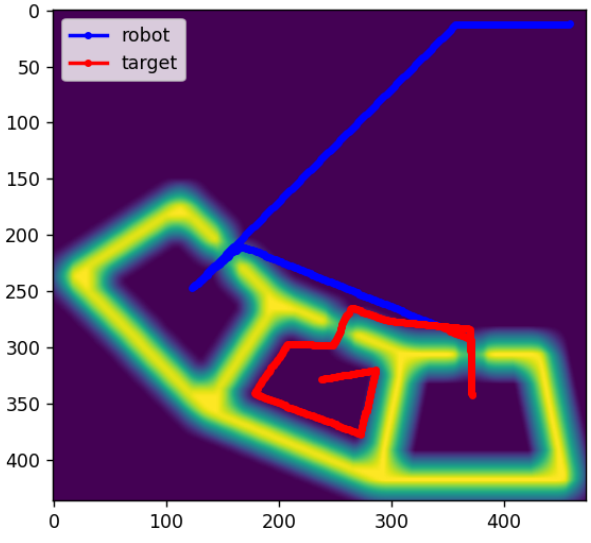

Figure 4: Map 4

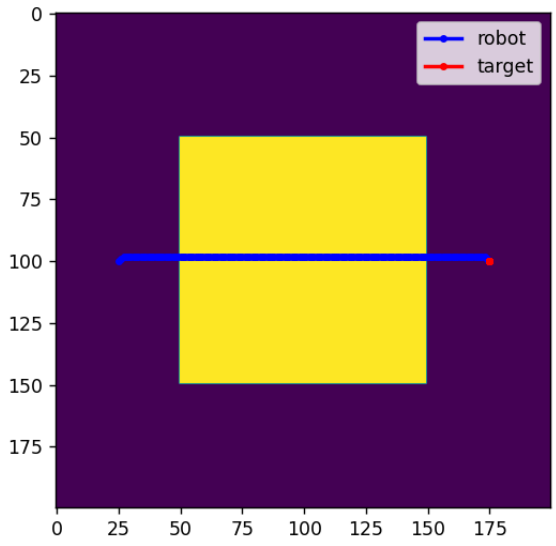

Figure 5: Map 5

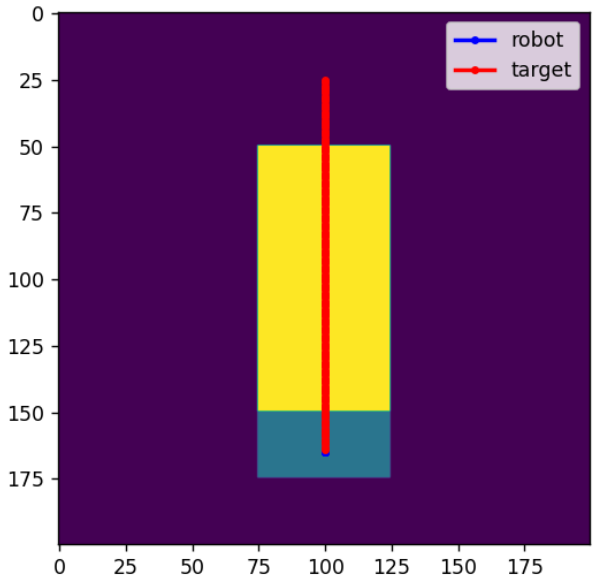

Figure 6: Map 6

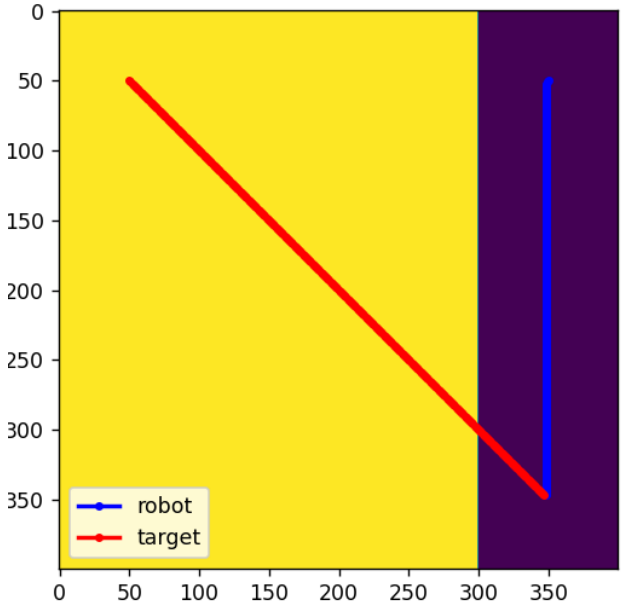

Figure 7: Map 7

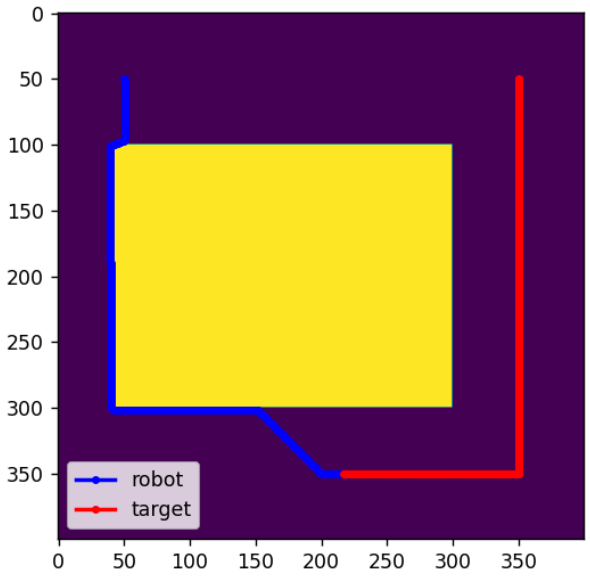

Figure 8: Map 8

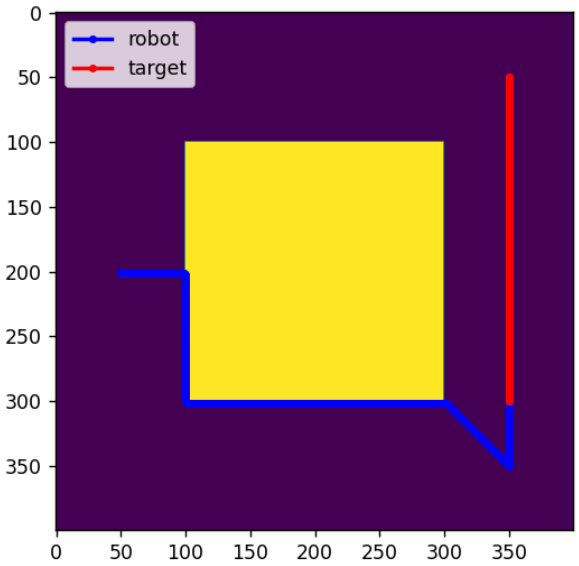

Figure 9: Map 9