# 16-350: Planning Techniques for Robotics
# Homework 1: Robot Chasing Target
# Due: Feb 14 (Fri), 11:59pm

David Seong
Spring 2025
Professor: Maxim Likhachev

February 14, 2025

## Contents

## 1 Introduction

This report details the approach taken to design and implement a planner that navigates a robot through a 2D grid to chase a target as fast as possible. Key components in the report are:

- A description of the planner algorithm, including data structures, heuristics, and efficiency techniques.
- Results for all maps including the result image, execution time, number of moves, and the overall path cost.

## 2 Planner Algorithm

The implemented planner uses a weighted backwards A* algorithm with an 8-connected grid to focus on the guarantee of capture and minimized path to target's goal position. The planner will navigate the robot to the end of the target's trajectory and trace back until the target's been caught.

### 2.1 Heuristic Map function

The `heuristic_map()` function uses the backwards A* algorithm to start search from the end goal position and searches the distance to all the grids in the map

- **distances, visited, OPEN:** Once a valid start point, end goal, is found, the grid is initialized as 0 and saved to *distances*, *OPEN*. When a grid is visited, its coordinates are added to *visited*.
- **Search Bounds:** Added a maximum search radius that was calculated from the quarter of the average of the width and length of the map to optimize the search. Any grid that is below the minimum search bound or above the search bound is automatically excluded from the heuristic map.
- **Backwards A*:** Starting from the end goal the algorithm searches the 8 neighboring grids, and for a grid that is valid, within search bounds, and has not been visited, algorithm calculates the distance from the preceding cell and adds to *OPEN*. It then searches through the *OPEN* map and selects the lowest distance cell, moves to it, sets current distance to previous distance + 1, add current grid to *visited*, and restarts the algorithm. Through this process, distance from the end point to all grid in the search bound will be determined.

### 2.2 Planner Algorithm

- **Cached Heuristic Map:** The planner is initialized with the goalposeX, Y as the end of the target trajectory and the corresponding backwards A* heuristic map created using the *Heuristic Map* function. Heuristic map is then cached so values can be inquired from the map without remapping at every step.

- **Weighted A*:** Heuristic obtained from the heuristic map multiplied by the half of the map's collision threshold is scaled down by a factor of 1.2 to be able to traverse through obstacles when required to reach the target but avoid it when not necessary. Weighted A*'s implementation was necessary to complete maps such as map5.txt where obstacle areas needed to be traversed to catch the target in time.
- **On Trajectory!:** Once the robot reaches a part of target's trajectory, the robot either maintains its position or backtracks the trajectory to meet the target to minimize the catching time of the target at the price of increased path cost.
- **Forward A*:** The planner also iterates through the 8 neighboring grids as long as they are valid, not visited, exist in the heuristic map, to find the optimal successor grid. For a grid, their weighted backwards A* heuristic (*epsilon * backwards A* heuristic* is accompanied by the grid's cost as well as the *euclidean heuristic*, to break any possible ties in the total cost.

### 2.3 Data Structures and Efficiency

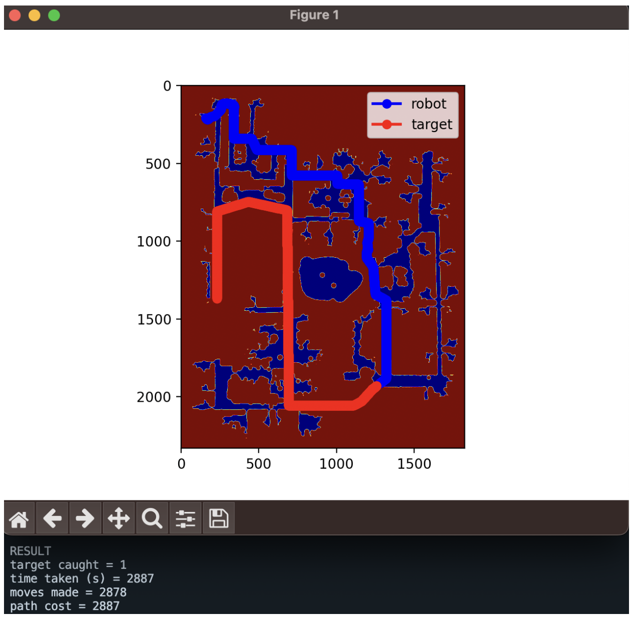
- **Priority Queue:** Used to select the grid cell with the minimum cost (*OPEN* list).
- **Maps, pairs:** `std::map` stores heuristic distances while `std::pairs` tracks visited positions.
- **Static Caching:** The heuristic map is computed once and then reused to improve efficiency.

## 3 Experimental Results

For each tested map, the following information was recorded:

- **Object Caught:** Yes/No.
- **Time Taken:** Execution time for the test.
- **Number of Moves:** Total moves made by the robot.
- **Path Cost:** The cumulative cost of the traversed path.

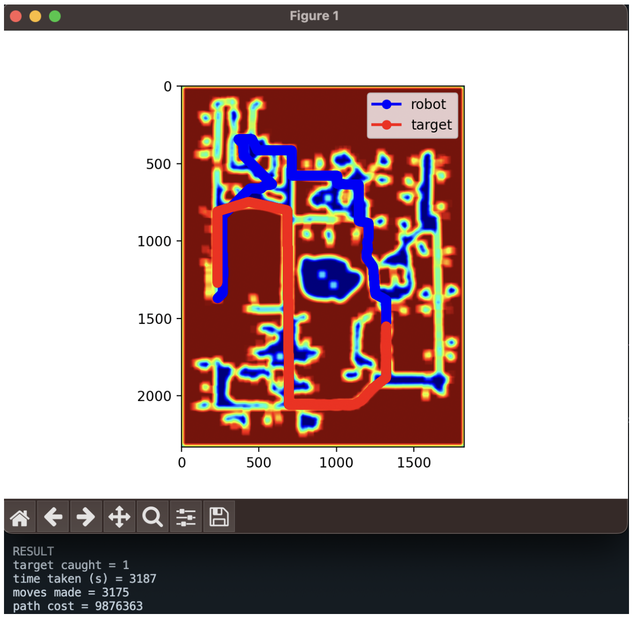
## 3.1 Map 1



```
RESULT
target caught = 1
time taken (s) = 2887
moves made = 2878
path cost = 2887
```

Object caught: Yes — Time taken: 2887 sec — Number of moves: 2878 — Path cost: 2887

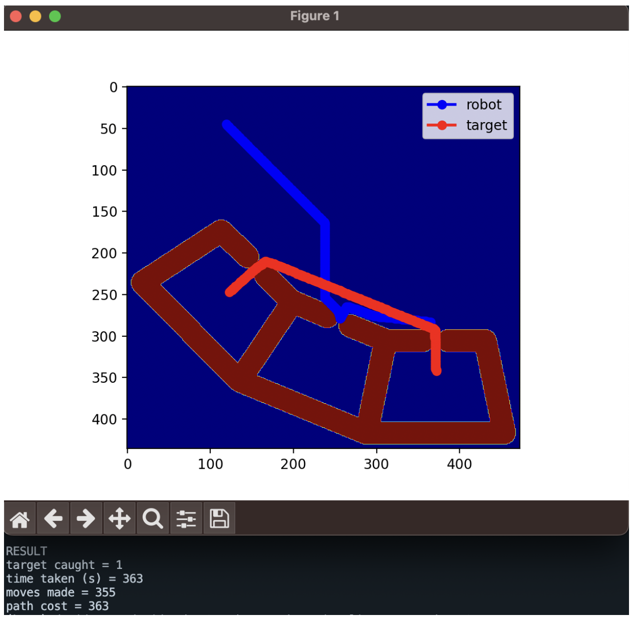
## 3.2 Map 2



```
RESULT
target caught = 1
time taken (s) = 3187
moves made = 3175
path cost = 9876363
```

Object caught: Yes — Time taken: 3187 sec — Number of moves: 3175 — Path cost: 9876363

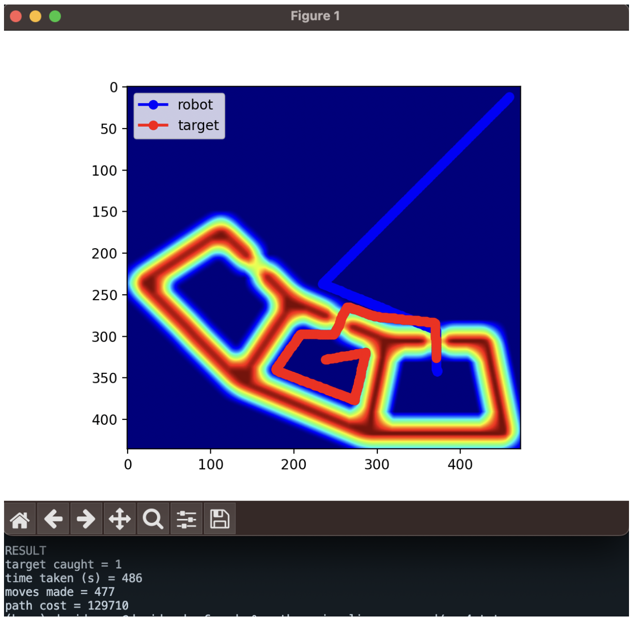
## 3.3 Map 3



```
RESULT
target caught = 1
time taken (s) = 363
moves made = 355
path cost = 363
```

Object caught: Yes — Time taken: 0363 sec — Number of moves: 355 — Path cost: 363

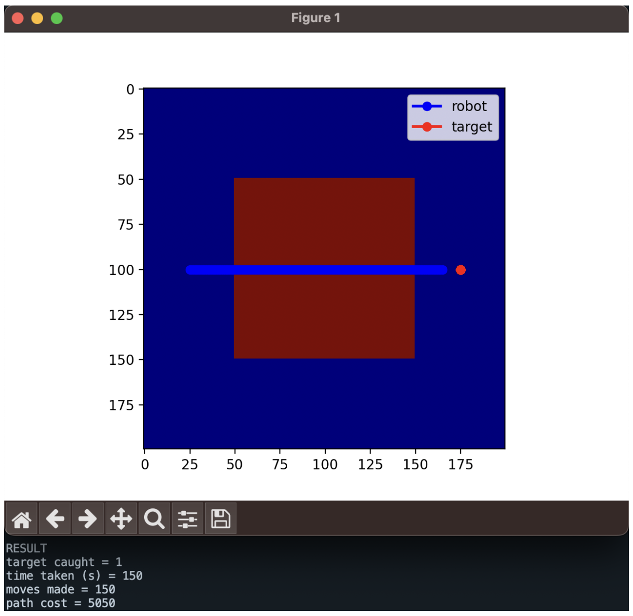
## 3.4 Map 4



```
RESULT
target caught = 1
time taken (s) = 486
moves made = 477
path cost = 129710
```

Object caught: Yes — Time taken: 486 sec — Number of moves: 477 — Path cost: 129710

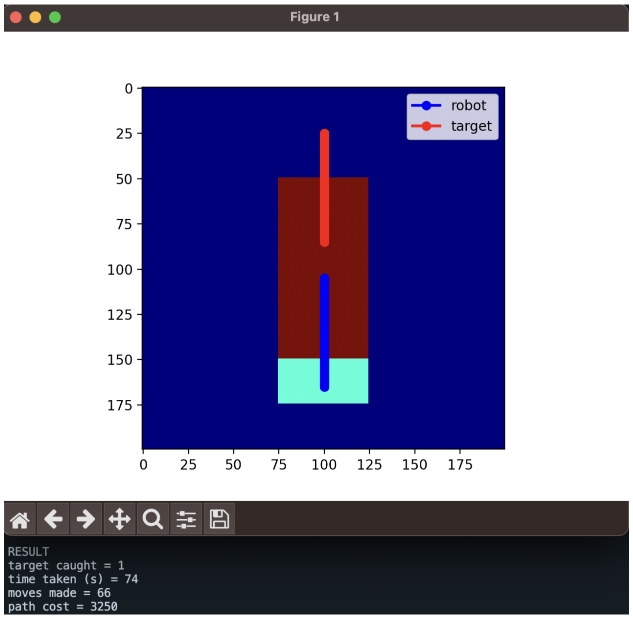
## 3.5 Map 5



```
RESULT
target caught = 1
time taken (s) = 150
moves made = 150
path cost = 5050
```

Object caught: Yes — Time taken: 150 sec — Number of moves: 150 — Path cost: 5050

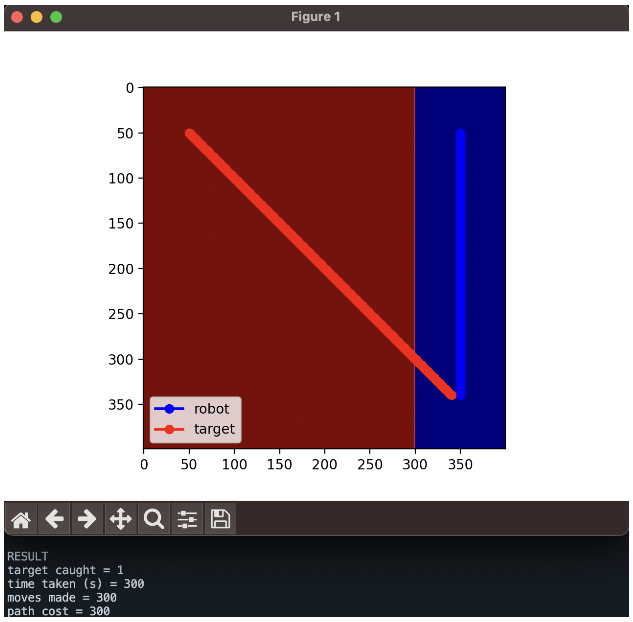
## 3.6 Map 6



```
RESULT
target caught = 1
time taken (s) = 74
moves made = 66
path cost = 3250
```

Object caught: Yes — Time taken: 74 sec — Number of moves: 66 — Path cost: 3250

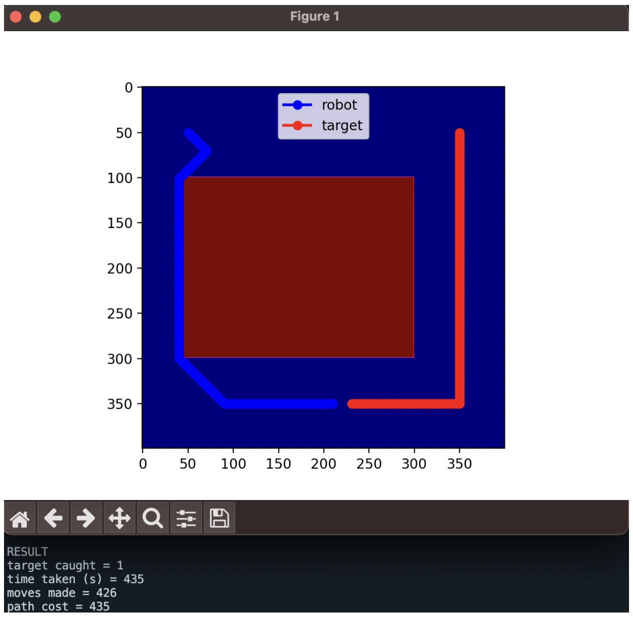
## 3.7 Map 7



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

Object caught: Yes — Time taken: 300 sec — Number of moves: 300 — Path cost: 300

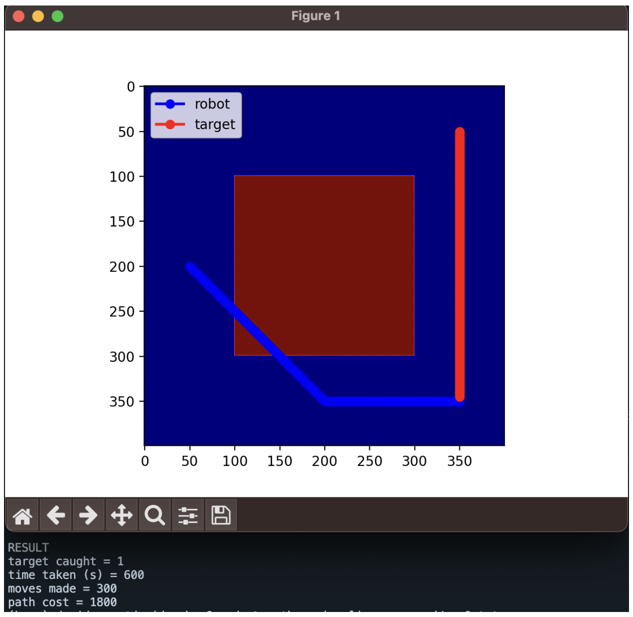
## 3.8 Map 8



```
RESULT
target caught = 1
time taken (s) = 435
moves made = 426
path cost = 435
```

Object caught: Yes — Time taken: 435 sec — Number of moves: 426 — Path cost: 435

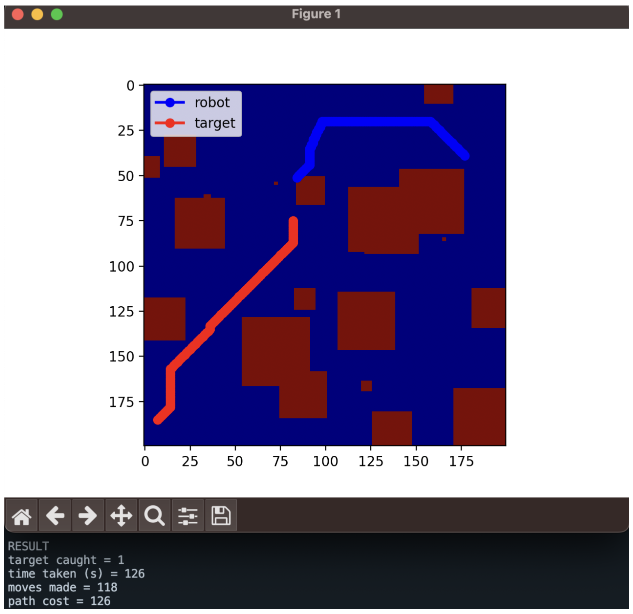
## 3.9 Map 9



```
RESULT
target caught = 1
time taken (s) = 600
moves made = 300
path cost = 1800
```

Object caught: Yes — Time taken: 600 sec — Number of moves: 300 — Path cost: 1800

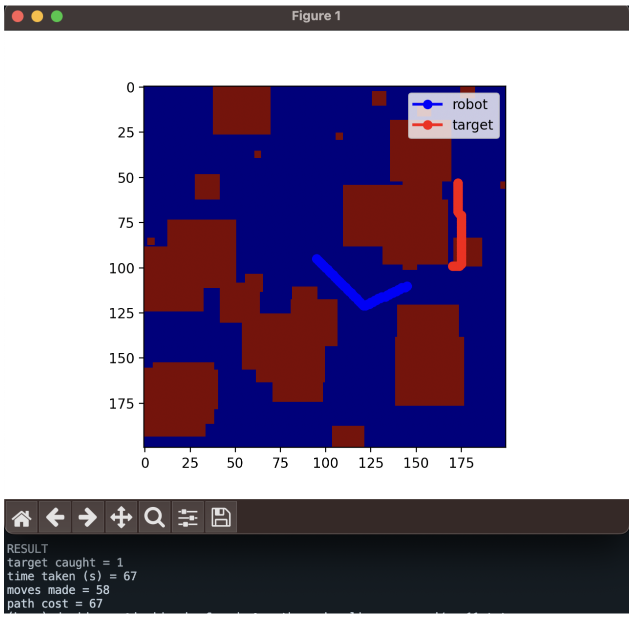
## 3.10 Map 10



```
RESULT
target caught = 1
time taken (s) = 126
moves made = 118
path cost = 126
```

Object caught: Yes — Time taken: 126 sec — Number of moves: 118 — Path cost: 126

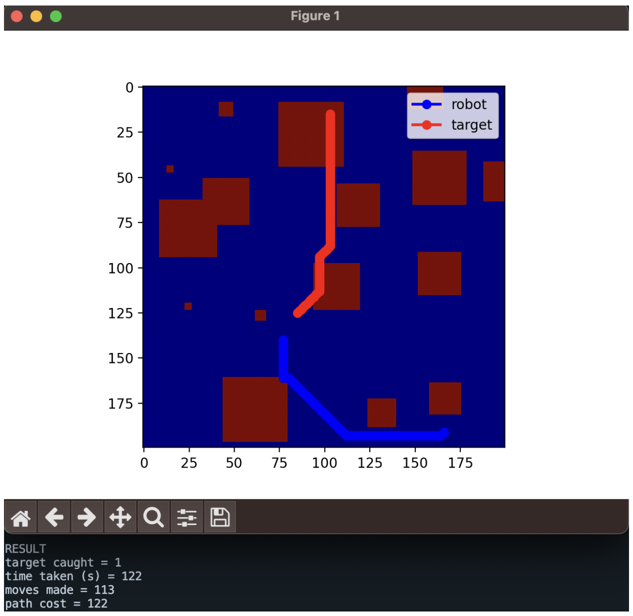
## 3.11 Map 11



```
RESULT
target caught = 1
time taken (s) = 67
moves made = 58
path cost = 67
```

Object caught: Yes — Time taken: 67 sec — Number of moves: 58 — Path cost: 67

## 3.12 Map 12



```
RESULT
target caught = 1
time taken (s) = 122
moves made = 113
path cost = 122
```

Object caught: Yes — Time taken: 122 sec — Number of moves: 113 — Path cost: 112

## 4 Compilation Instructions

The code was compiled and executed using the following commands:

```
g++ -std=c++17 runtest.cpp planner.cpp
./a.out undergrad/map5.txt
python visualize.py undergrad/map5.txt
```

## 5 Conclusion

In summary, the planner efficiently computes a path for the robot by combining a weighted backward A*, euclidean heuristics along with a forward A* algorithm. The experimental results indicate that the planner successfully navigates through all map configurations. One optimization method I was in the process of implementing was, instead of targeting the end of the trajectory, targeting various different points in the trajectory and only move on to the next checkpoint if goal seems unreachable in the given time. Another way to optimize the planner is to use the node data structure to more effectively calculate the cost and path from the robot's current position to the target and backtrack to determine which of the 8 grid to move on to next.