16-782: Planning and Decision Making in Robotics
Homework 1: Catching a Moving Target
Due: September 22 (Mon), 11:59pm

Professor: Maxim Likhachev *Fall 2025 TA: Alvin Zou, Barath Satheeshkumar*

1 Problem Description

In this problem set, you'll develop and implement a planning algorithm to catch a moving target within a 2D grid-world. The grid-world is 8-connected, allowing the robot and agent to move up to one unit in any direction—horizontally, vertically, or diagonally—per time step. Your planner will be given the robot's starting position and the target's trajectory, represented as a sequence of positions (e.g., [(5, 6),(5, 7),(6, 6)]). Additionally, the planner will be provided with a costmap, which assigns a non-negative integer cost to each cell, and a collision threshold. Any cell with a cost equal to or exceeding the collision threshold is considered an obstacle, preventing the robot from visiting it. The largest grid you'll encounter is around 2000 $\times$ 2000 cells. **Your task is to develop a planner that enables the robot to catch the target while incurring as little cost as it can.**

1.1 Code:

Your code is within the folder `code`, where you will have `C++` files (in the directories `src` and `include`) as well as a `maps` directory and a visualization script. The planner should reside in the `src/planner.cpp` file, and within it, the `planner` function must output a single robot move. Currently, the file contains a greedy planner that always moves the robot in the direction that decreases the distance between the robot and the target. The signature of the `planner` function (inside `src/planner.cpp`) is as follows:

```
static void planner(
     double *map,
     int collision_thresh,
     int x_size,
     int y_size,
     int robotposeX,
     int robotposeY,
     int target_steps,
     double *target_traj,
     int targetposeX,
     int targetposeY,
     int curr_time,
     double *action_ptr
     )
{}
```

1.2 Inputs:

Each cell in the `map` of size `(x_size, y_size)` is associated with the cost of moving through it (positive integer). Note that if the robot stays in the same cell c for T time steps, then it will incur a cost of $cost(c) \cdot T$. The cost of moving through cell `(x, y)` in the `map` should be accessed as:

`(int)map[GETMAPINDEX(x,y,x_size,y_size)]`.

If it is less than `collision_thresh`, then the cell `(x, y)` is free. Otherwise, it is an obstacle that the robot cannot traverse. Note that cell coordinates start with 1. In other words, `x` can range from `1` to `x_size`. The target's trajectory `targe_traj` of size `target_steps` is a sequence of target positions (for example: (2,3), (2,4), (3,4)). At the current time step (`curr_time`), the current robot pose is given by `(robotposeX, robotposeY)` and the current target pose is given by `(targetposeX, targetposeY)`. The target will also be moving on the 8-connected grid, at the speed of one step per second along its trajectory. Therefore, at the next second, the target will be at `(current_time + 1)`$^{\texttt{th}}$ step in its trajectory `target_traj`.

You are provided with a few maps. Target, robot and map cost information is specified in text files named `map*.txt`. Specifically, the format of the text file is:

1. The letter N followed by two comma separated integers on the next line (say N1 and N2 written as N1,N2). This is the map's size.
2. The letter C followed by an integer on the next line. This is the map's collision threshold.
3. The letter R followed by two comma separated integers on the next line. This is the starting position of the robot in the map.
4. The letter T followed by a sequence of two comma separated integers on each line. This is the trajectory of the moving object.
5. The letter M followed by N1 lines of N2 comma separated floating point values per line. This is the map.

The file `runtest.cpp` parses the text files, and calls your planner function (with these inputs) once per simulation step.

1.3 Outputs:

At every simulation step, the planner function should output the robot's next pose in the 2D vector `action_ptr`. The robot is allowed to move on an 8-connected grid. All the moves must be valid with respect to obstacles and map boundaries (see the current planner inside `planner.cpp` for how it tests the validity of the next robot pose).

The `runtest.cpp` script returns four values - a boolean specifying whether the object was caught, and three integers specifying the time taken to run the test, the number of moves made by the robot, and the cost of the path traversed by the robot.

1.4 Frequency of Moves:

The planner is supposed to produce the next move within 1 second. Within 1 second, the target also makes one move. If the planner takes longer than 1 second to plan, the target will have moved by a longer distance in the meantime. In other words, **if the planner takes K seconds (rounded up to the nearest integer) to plan the next move of the robot, then the target will move by K steps in the meantime.**

Note: After the last cell on its trajectory, the object disappears. So, if the given object's trajectory is of length 40, then at time step = 41 the object disappears and the robot can no longer catch it. This means that, **for a moving object trajectory that is T steps long, your planner has at most T seconds to find (and execute) a full solution.**

1.5 Execution:

The following are instructions for compiling the `C++` code. Instruction using CMake and `g++` are included. We encourage you to use CMake. Commands will differ between operation systems. The ones below are common in Unix-based systems (Ubuntu Linux, Mac, etc.).

1.5.1 Building with CMake

Open a terminal and navigate to the `code` directory.

```
>> mkdir build
```

```
>> cd build
```

To build in release mode:

```
>> cmake .. -DCMAKE_BUILD_TYPE=Release
```

To build in debug mode (needed when using debuggers and will run more slowly):

```
>> cmake .. -DCMAKE_BUILD_TYPE=Debug
```

Build the code with:

```
>> make
```

Run the code with:

```
>> ./run_test map<map_number>.txt
```

An example command would be `./run_test map9.txt`. The results of this run will be saved to the output directory.

1.5.2 Building with `g++`

Open a terminal and navigate to the `code` directory.

```
>> g++ src/runtest.cpp src/planner.cpp -o run_test
```

To run the planner:

```
>> ./run_test map<map_number>.txt
```

1.5.3 Visualization

To visualize the robot and target's trajectory, navigate to the `scripts` directory and run:

```
>> python visualizer.py ../maps/map<map_number>.txt
```

For example, to visualize the robot and target's trajectory for `map9.txt`, run:

```
>> python visualizer.py ../maps/map9.txt
```

Currently, the planner greedily moves towards the last position on the moving object's trajectory and waits there.

1.6 Submission:

You will submit this assignment through Gradescope. You must upload one ZIP file named `<andrewID>.zip`. This should contain:

1. A folder `code` that contains all code files, including but not limited to, the ones in the homework packet. If you add subfolders, your code should handle relative paths.
2. Your writeup in `<andrewID>.pdf`. This should contain a summary of your approach for solving this homework, the results for all maps (whether the object was caught, the time taken to run the test, number of moves made by the robot, and the cost of the path traversed by the robot), and instructions for how to compile your code.
   - For your planner summary, we want details about the algorithm you implemented, data structures used, heuristics used, any efficiency tricks, memory management details etc. Basically any information you think would help us understand what you have done and gauge the quality of your homework submission.
   - Include plots of the maps overlaid with the object and solved robot trajectories. Please **do not** include the map text files in your submission.

1.7 Grading:

The grade will depend on two things:

1. How well-founded the approach is. In other words, can it guarantee completeness (to catch a target, if one exists), can it provide sub-optimality or optimality guarantees on the paths it produces, can it scale to large environments?
2. How much cost the robot incurs while catching the target.

Note: To grade your homework and to evaluate the performance of your planner, we may use different maps than the ones provided in the directory. We can only promise that the sizes of these maps will not exceed 2000 $\times$ 2000 cells.

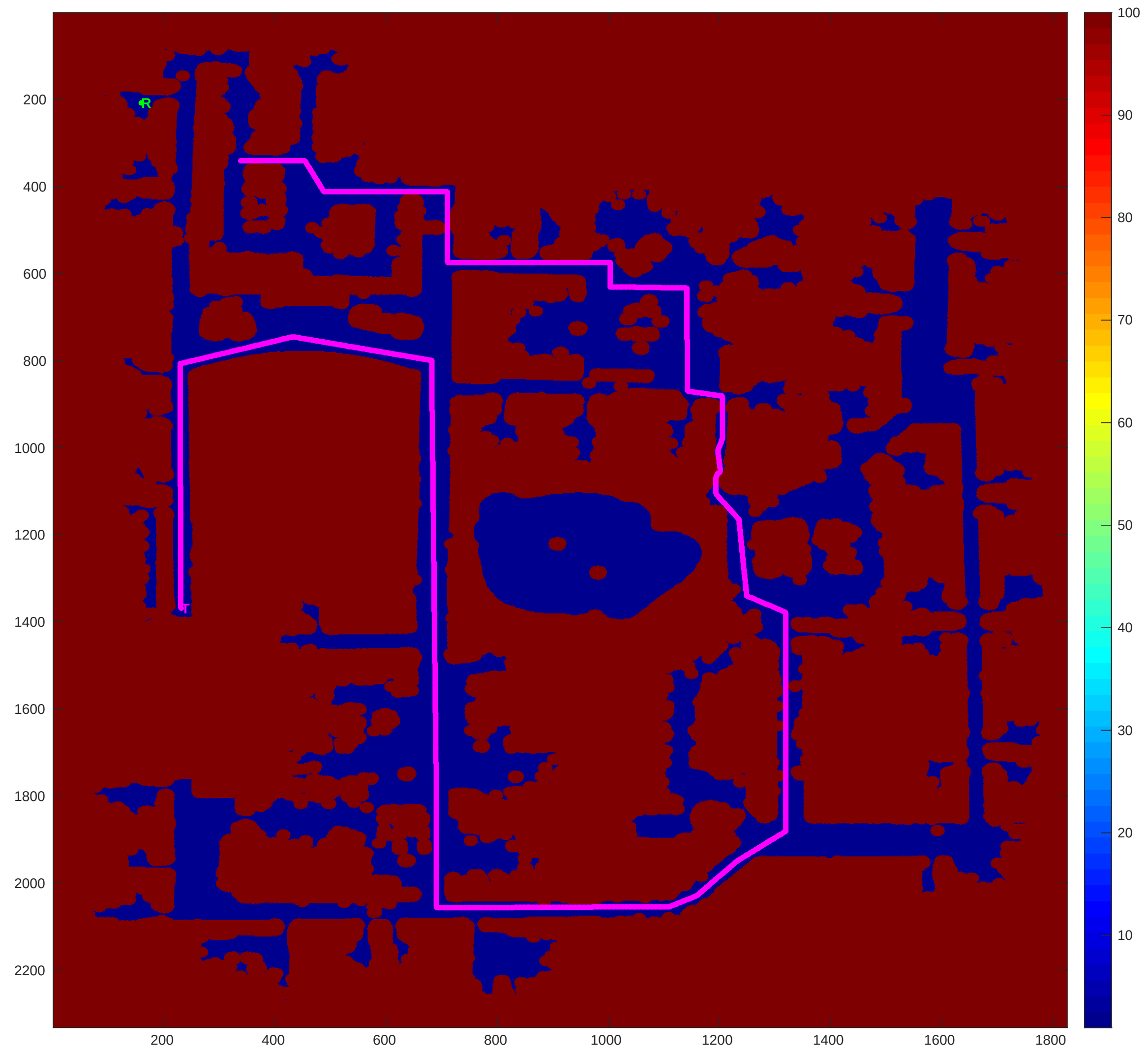

Figure 1: Image of information in `map1.txt`. The green R marks the starting position of the robot, the magenta T marks the starting position of the target, the magenta line is the target's trajectory. Blue cells have cost 1, red cells have cost 100, collision threshold is 100.

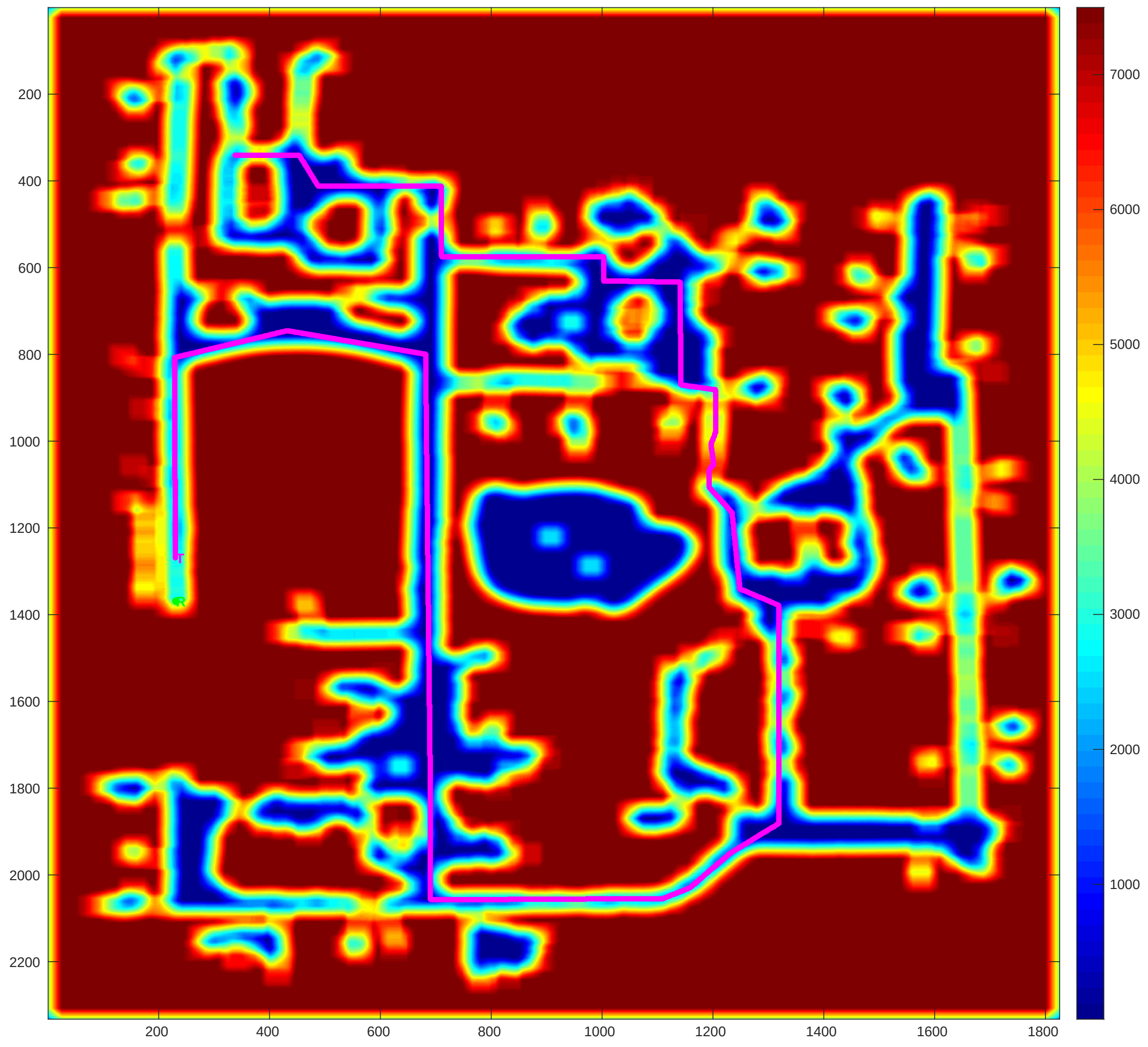

Figure 2: Image of information in `map2.txt`. The green R marks the starting position of the robot (directly below the starting position of the target), the magenta T marks the starting position of the target, the magenta line is the target's trajectory. Cells have cost between 1 and 7497, inclusive. Collision threshold is 6500.

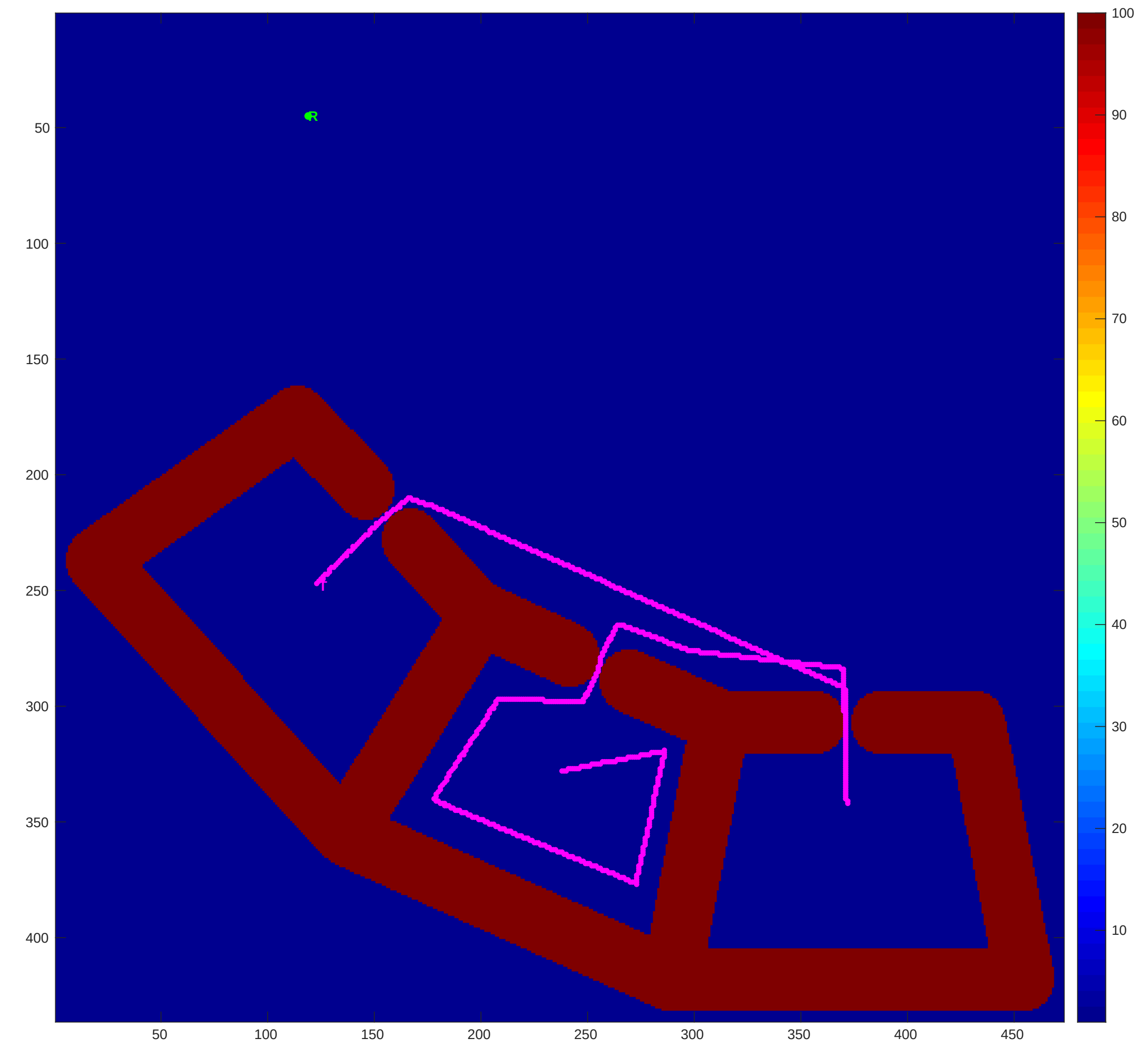

Figure 3: Image of information in `map3.txt`. The green R marks the starting position of the robot, the magenta T marks the starting position of the target, the magenta line is the target's trajectory. Blue cells have cost 1, red cells have cost 100, collision threshold is 100.

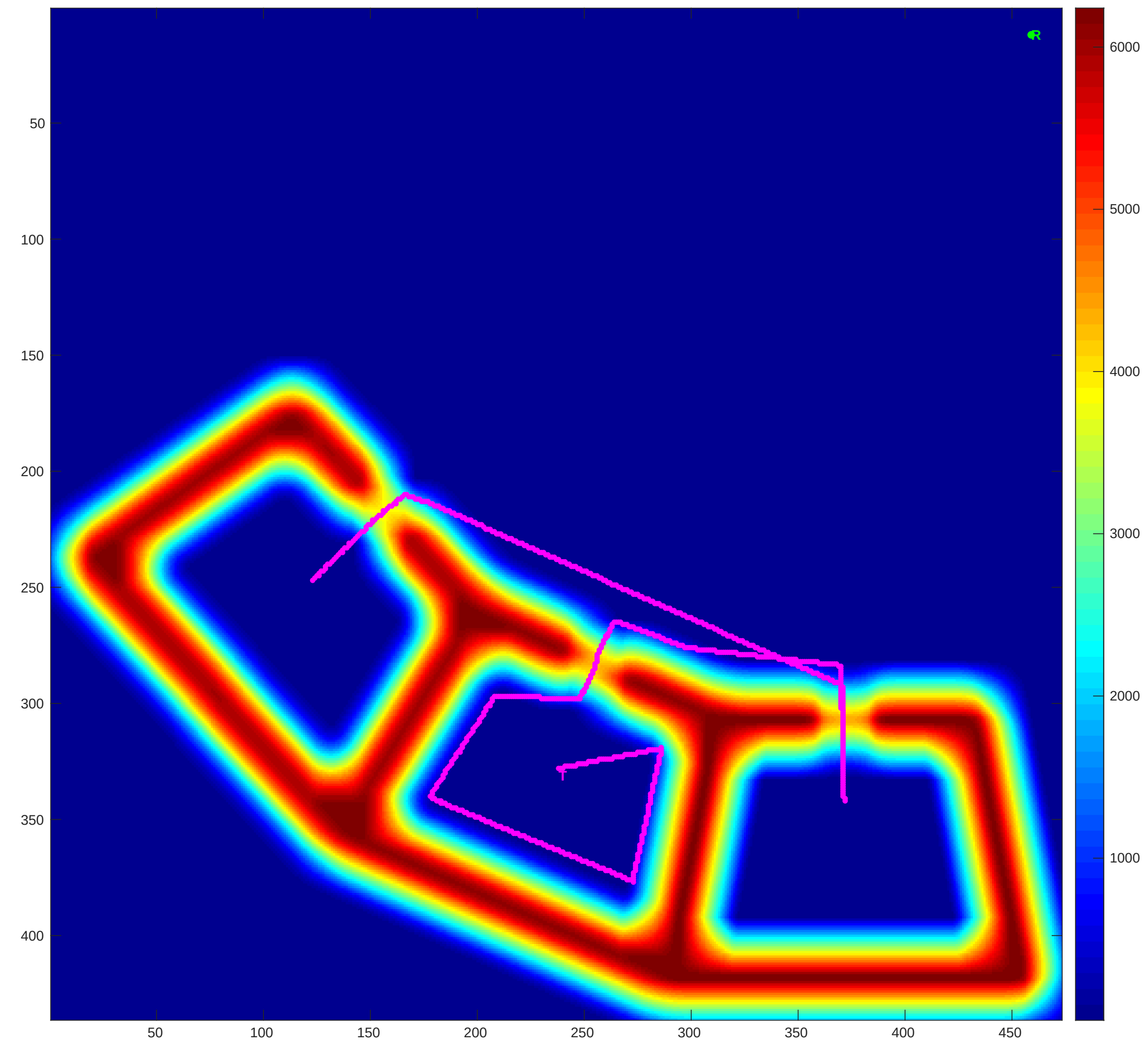

Figure 4: Image of information in `map4.txt`. The green R marks the starting position of the robot, the magenta T marks the starting position of the target, the magenta line is the target's trajectory. Cells have cost between 1 and 6240, inclusive. Collision threshold is 5000.