COMP0037 2024 / 2025 Robotic Systems

Lab 02: Graph-Based Search

COMP0037 Teaching Team

January 23, 2025

Overview

In this lab, we will explore some of the ideas and techniques required for graph-based search for a robot. It covers material from Lecture 03 on discrete planning, graph-based search and dynamic programming. It will also introduce you to the software which is used for CW1.

Installation Instructions

This code uses the Zelle graphics module to manage the window for graphics output. The module is included with the code. However, it uses tkinter, which might not be installed by default. The requirements file `requirements-1.txt` should include this extra dependency.

Preliminaries

The activities in this section focus on getting familiar with the basic system.

1. The file `run_breadth_first_empty_space.py` contains the launcher code to plan a path in an open space. All the activities in this question will ask you to use it.

    a. Run the script and confirm that you get an output which looks like the one in Figure 1.a.

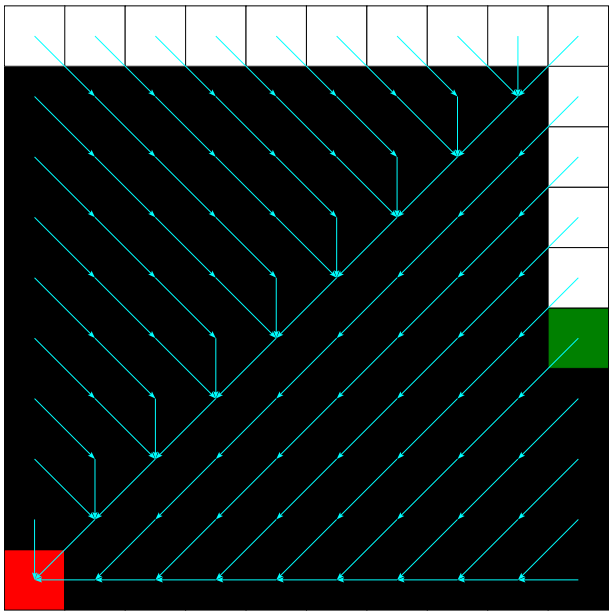

Figure 1: The search grid you should see from running Q1a.

b. Modify the script to plan paths between the following destinations:

Start	Goal
(1,1)	(3,5)
(1,7)	(8,4)

Do you notice any funny cases with the paths generated?

c. Modify `run_breadth_first_empty_space.py` to make the map 30×30 cells. Experiment with planning a few paths in this scenario. If the window is too large for your screen, you can change the height of the displayed window before the planner is called to plan the first path. The code shows you how to do this.

d. Add obstacles to the grid created in `run_breadth_first_empty_space.py`. What does the algorithm do if the goal becomes unreachable because of those obstacles?

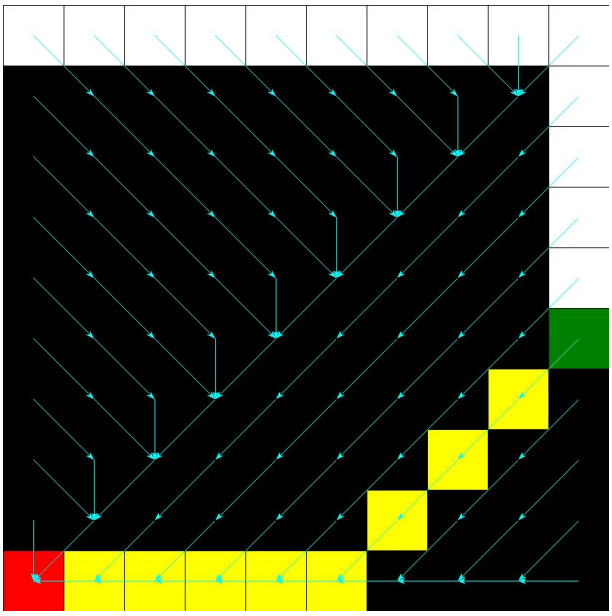

Figure 2: An example of an extracted path (yellow) from the start (green) to the goal (red) in Q2a.

2. In this task, we will add capabilities to `planner_base.py` to extract the path and compute the cost of that path.

   a. Complete the implementation of the method `extract_path` to compute the path from the specified cell to the start. An example of the output you should see is in Figure 2.

   b. Extend `extract_path` to compute the number of cells on the path and the length of the path. The length of the path is computed by adding up the distances between each cell-to-cell transition on the path. Check your results on a few different paths.

3. This task will look at some slightly less-trivial examples. It uses the script `run_breadth_first.py` which runs the breadth first algorithm in a slightly more complicated scenario with a single wall.

   a. Run the example in the original search order and note where the suboptimalities lie.

   b. Modify the method `next_cells_to_be_visited` in `planner_base.py` so that the cells are visited in a different sequence. Investigate the impact of changing the search order on the computed path.

   c. Change the design of the walls to see how these interact with your choice of order in `run_breadth_first.py`.

4. This final task asks you to experiment with a planner which uses different types.

   a. Complete the implementation of the `GreedyShortestDistancePlanner` in `greedy_shortest_distance_planner.py`. The priority, d, is the Euclidean distance from the cell to the goal. Compare its performance with the breadth first planner for the environments provided.

   You will need to use a priority queue. I personally use `PriorityQueue`, but you can use `heapq` directly. See this guide for more details.

   b. Somebody thinks it would be a great idea to change the priority to $10^6 - d$. What happens in this case?