COMP0037 2021 / 2022 Robotic Systems

Lab 02: Graph-Based Search

COMP0037 Teaching Team

January 21, 2022

Overview

In this lab, we will explore some of the ideas and techniques required for graph-based search for a robot. This lab covers the first part of Lecture 04 (up to and including Part 07) and includes breadth first search.

Stuff to get them to do:

Implement L-stage additive cost

Instrument code to compute path length

Try different types of queues

Try different search orders

Get them to implement Dijkstra without explicitly stating it.

Installation Instructions

This code uses the Zelle graphics module to manage the window for graphicsl output. The module is included with the code. However, it uses tkinter, which might not be installed by default.

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, and we will start by using it.

   a. Modify `run_breadth_first_empty_space.py`, to plan paths between the following destinations:

Start	Goal
(10,10)	(10,0)
(0,0)	(20,10)

   b. 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, reduce the valute of `maximumGridDrawerWindowHeightInPixels` in the planner before planning a path.

   c. 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?

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 `extractPath` to compute the path from the specified cell to the start.

   b. The l-stage additive cost of moving between two cells is the Euclidean distance between the centre of those cells. Implement this cost function in the method `computeLStageAdditiveCost`.

   c. Extend `extractPath` to compute the length of the extracted path. Modify `extractPath` to print out details on the extracted path including: the travel length along the path, and the number of cells on the path.

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 `nextCellsToBeVisited` 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`. 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.