UNIVERSITÀ DEGLI STUDI DI GENOVA

DIBRIS

DEPARTMENT OF COMPUTER SCIENCE AND TECHNOLOGY,
BIOENGINEERING, ROBOTICS AND SYSTEM ENGINEERING

ARTIFICIAL INTELLIGENCE FOR ROBOTICS II

Second Assignment

Task and Motion Planning

Author:

Manuel Delucchi, Claudio Demaria,
Gianluca Galvagni, Emanuele
Giordano, Enrico Piacenti

Professor:

Fulvio Mastrogiovanni

Professor's assistance:

Antony Thomas

July 2, 2023

1 Introduction

In *AI*, *PDDL* is a *planning language* utilized for representing plans and actions. However, it has limitations in expressing complex equations. To overcome this challenge, *Semantic Attachment* in *PDDL* integrates a generic planner with a specialized advisor, enabling advanced mathematical computations by evaluating fluents using external functions.
In the context of this assignment, our focus is on *Task Motion Planning (TMP)* problems, which involve the navigation and manipulation of objects. Specifically, we explore the application of Semantic Attachment in *PDDL* as a tailored solution for addressing *TMP* problems.

This paper presents a *task and motion planning framework* designed to achieve the shortest path assignment collection in a *2D* simulation environment. The environment has dimensions of *6m x 6m*, with the submission desk located at coordinates *(3, 0)*. The robot starts at position *(0, 0)*, and the four regions, each measuring *1m x 1m*, are positioned at the corners of the environment. Each region is associated with a single waypoint *(x,y)*.

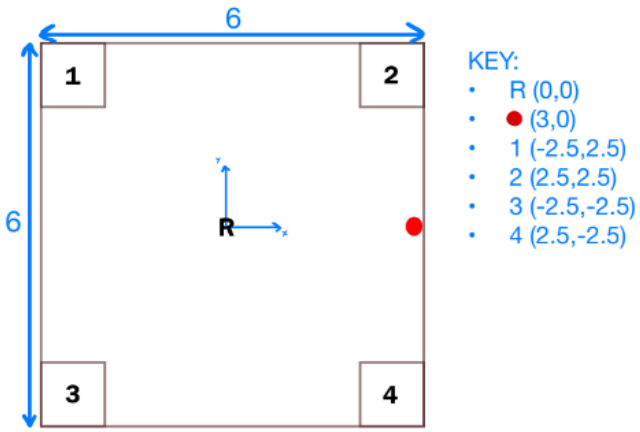

Figure 1: Illustration of the 2D environment where the robot located at start in R has the ability to navigate to regions 1-4 as well as the designated location highlighted in red.

To facilitate efficient path planning, the framework uses a roadmap constructed from a combination of six predefined waypoints and 24 randomly sampled waypoints located outside the regions. These waypoints are interconnected, forming edges and creating a comprehensive network. The number of connections, denoted as *k*, ensures each waypoint is associated with a maximum of *k* other waypoints, optimizing the planning process.
The aim is to *minimize the motion cost* while collecting two assignment reports and delivering them to the submission desk. The motion cost is determined by the total length of the robot's path, which is computed by summing the *Euclidean distances* between the traversed waypoints during the assignment collection process.
To tackle the challenge of achieving the goal of shortest path assignment collection, we utilized the *popf-tif* planner, allowing us to link *C++* code with *PDDL*, and made various modifications to the domain, problem, and external modules. These modifications were specifically targeted at optimizing the planning process to identify the most efficient path for the robot
The proposed task and motion planning approach combine the benefits of a well-constructed roadmap, waypoint connections, and optimized path planning. This framework provides an efficient solution for achieving the shortest path assignment collection within the given *2D* environment.

2 Implementation

We focused primarily on the *VisitSolver* class in *C++* and the *PDDL* files during our work.

To begin with, we implemented a random waypoint generator function, denoted as *"randWaypointGenerator()"*. This is a function that, additionally to the already existing 6 waypoints, generates 24 new random waypoints. The process involves opening the *"waypoint.txt"* file and clearing its contents each time to avoid manual intervention. The function then writes the 6 known waypoints and generates 24 new random waypoints using a uniform distribution between -3.0 and 3.0. It's worth noting that the orientation theta is always set to 0 since our work pertains to a *2D* simulation environment and as it is not necessary for calculating the Euclidean distance.

Next, we proceeded by creating a graph of waypoints, connecting each waypoint to its *k* nearest neighbors. This process was accomplished using the *"buildGraph()"* function. The function constructs a graph by establishing connections between each waypoint and a maximum of *k* other waypoints. The user can specify the value of *k*, ranging from 5 to 30.
To build the graph, we computed the Euclidean distances between the waypoints and populated a distance matrix accordingly. The Euclidean distance formula used was: $dist = \sqrt{(x_f - x_i)^2 + (y_f - y_i)^2}$, since we assumed a *2D* scenario with x and y coordinates. However, we ensured that the self-connections were excluded by assigning a high value to the diagonal elements of the matrix. This precaution prevents the minimum distance calculation from being influenced by the zero distance between identical waypoints.
For the representation of the graph, we utilized an *adjacency matrix*. In simple terms, an adjacency matrix is a square matrix where the rows and columns correspond to the waypoints. If there is a connection between two waypoints, the corresponding matrix entry is set to 1; otherwise, it is set to 0.

The last step consists in using the *Dijkstra algorithm* that is a popular algorithm used to find the shortest path between two nodes in a graph. The Dijkstra algorithm guarantees that the first time a node is visited, its distance from the starting node is the shortest possible distance. By iteratively selecting the node with the smallest tentative distance, the algorithm explores all possible paths from the starting node to the destination node and determines the shortest one.
Using this algorithm, you can compute the shortest path between two regions by representing the regions as nodes and the connections between them as edges in a graph. The Dijkstra algorithm then efficiently finds the shortest path between these two regions based on the weights of the edges. For a step-by-step understanding of this algorithm, you can refer to the *"compute_path()"* function implemented inside the VisitSolver.cpp class. This function provides further explanation and guidance throughout the code with associated comments. Alternatively, you can consult the following website at this link: Dijkstra algorithm.

As mentioned earlier, the motion cost is determined by the total length of the robot's path. In the *"compute_path()"* function, the total cost is computed and stored in the *act_cost* variable. Subsequently, this value is then passed to the *PDDL* domain file, allowing the planner to access it. By using this information, the planner selects actions to optimize the overall path.

Link to our GitHub repository → Here you can also find how to *install and run* the *popf-tif planner*.

3 Results

Examples of how the script works in the terminal are shown below:

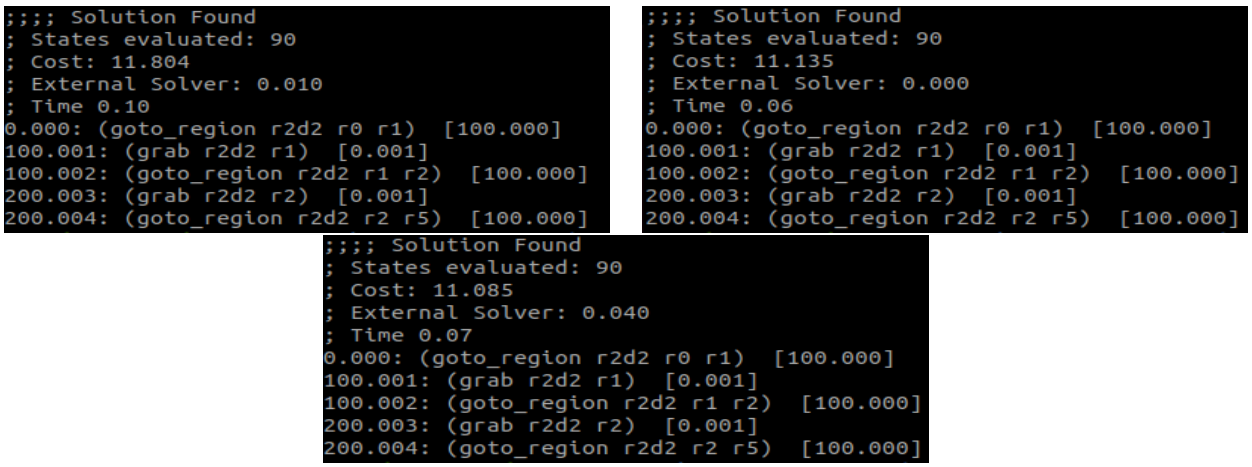

Figure 2: "The top left figure showcases the results for k = 5, the top right figure showcases the results for k = 18 and the bottom one showcases the results for k = 30."

By increasing the value of *k*, we improve each node's connectedness, thereby expanding the number of available pathways between nodes. This improvement in connectivity ultimately leads to better performance in identifying low-cost pathways. It's important to consider that increasing *k* requires additional computational resources for graph generation and route determination. However, since our project is relatively small and doesn't involve a large number of nodes or complex graphs, the increase in computational resources required by raising the value of *k* may not be a significant concern.