HW 1—Exact motion planning

1 General guideline

This homework will cover the topic of exact motion planning. It includes one quick "dry" exercise (2) and one "wet" programming exercise (3). Writeups must be submitted as a PDF. LaTeX is preferred, but other typesetting methods are acceptable. Code for the programming component must be submitted in a zip archive. Plots generated as part of the programming component should be included in the writeup. The homework can be submitted in pairs.

Submission date is Thursday 25.11 end of day.

2 Properties of Minkowski Sums and Euler's theorem (15 points)

- Given sets A, B and C formally **prove** that $A \oplus (B \cup C) = (A \oplus B) \cup (A \oplus C)$
- **What** is the Minkowski sum (what geometric object and what can you say about it) of (i) Two points? (ii) A point and a line? (iii) Two lines segments (think of all possible cases)? (iv) Two Disks?
- Recall that for the proof of Lemma. 6.2 (complexity of a trapezoidal map) we used the property that in a planar graph we have that

$$E \leq 3V - 6. \tag{1}$$

  Here E and V are the number of edges and vertices in a planar graph, respectively. **Prove** Eq. 1 (you can assume that $V \geq 3$).

3 Exact Motion Planning for a Diamond-Shaped Robot (85 points)

In this section you will be required to implement in Python a motion-planning algorithm to efficiently compute the motion of an axis-aligned square-shaped robot rotated by 45° translating amidst simple convex obstacles.

Specifically, we will preprocess an environment that contains obstacles which are either squares or triangles. This will allow us to efficiently answer queries in the online phase.

3.1 Code Overview

The starter code is written in Python and depends on NumPy and Matplotlib. This section gives a brief overview.

- `HW1.py` - Contains the main function. Note (below) the command-line arguments that you can provide. This file contains empty functions (`get_minkowsky_sum` and `get_visibility_graph`) that you will have to fill as well as variables (`shortest_path` and `cost` in line 92) you will have to populate. See the next sections for details.
- `Plotter.py` - Contains visualization functions provided to you.
- `robot` - Contains a description of the robot. Specifically, this is represented as the tuple (x, y, d) where (x, y) denotes the coordinates of the center of the robot, also serving as its starting point, and d denotes the distance from the center to its vertices.
- `query` - Contains a point (x, y) that denotes the target coordinates for the center of the robot.
- `obstacles` - Contains the environment obstacles. Each line represents a single obstacle, and it holds the coordinates of its vertices.

The only library that is not built-in in python is "Shapely". It can be installed using pip (i.e., type `pip install shapely`).

To run the code, simply run `python3 HW1.py robot obstacles query`

For each of the following sections, use the input files (`robot`, `query` and `obstacles`) provided to you and generate one more instance. Show results for both settings (the one provided and the one you generated).

3.2 Preprocessing phase (1)—constructing the C-space

In the first step, we will construct the C-space of our rotated square-shaped robot $\mathcal{R}$. This will be done by computing the Minkowski sum of $\mathcal{R}$ with each obstacle O. **Visualize** the C-space and **describe** the computational complexity of your implementation.

Specifically, here you need to implement `get_minkowsky_sum` which returns a list polygons representing the C-Space obstacles.

In your implementation, you can assume that all obstacles are convex. How can non-convex obstacles affect the results? Why? When? Support your claim with examples.

3.3 Preprocessing phase (2)—building the visibility diagram

In the second step, we will build a visibility graph over the C-space we constructed in the first part of the preprocessing phase. **Visualize** the visibility graph and **describe** the computational complexity of your implementation.

Specifically, here you need to implement `get_visibility_graph` which updates the variable `lines` (line 90) that holds a list of lines (`LineString`) connecting the vertices of the C-Space obstacles (if there is no obstacle intersecting this line).

3.4 Query phase—computing shortest paths

In the query phase we will need to update the visibility graph to include the query and implement Dijkstra's algorithm to compute a shortest path between the start and the goal. For each one of the queries, **Visualize** the solution obtained and **describe** the computational complexity of your implementation.

Specifically, `lines` (in line 90) should hold a list of line segments (`LineString`) connecting the vertices of the C-Space obstacles (if the segment is collision-free) and the start/end position of the robot. `shortest_path` and `cost` should hold the shortest path for the robot from the starting position to the end position, and its cost.