Exact Methods in Path Planning

1 Introduction

In this first homework, we explore exact computational methods for motion-planning.

2 Properties of Minkowski Sums and Euler's theorem

2.1 Given sets A,B and C prove that $A \oplus (B \cup C) = (A \oplus B) \cup (A \oplus C)$

In geometry, the Minkowski sum of two sets of position vectors A and B in Euclidean space is formed by adding each vector in A to each vector in B, i.e.:

$$A \oplus B = \{a + b \mid a \in A, b \in B\}$$

Direction 1:

$Let\, p \in A, q \in (B \cup C)$

$q \in (B \cup C) \rightarrow q \in B\ or\ q \in C$

a. $q \in B \rightarrow p + q \in (A \oplus B)$

b. $q \in C \rightarrow p + q \in (A \oplus C)$

To summarize: $\mathbf{p \in A, q \in (B \cup C) \rightarrow p + q \in (A \oplus B) \cup (A \oplus C) \rightarrow A \oplus (B \cup C) \subseteq (A \oplus B) \cup (A \oplus C)}$

Direction 2:

$Let\, p + q \in (A \oplus B) \cup (A \oplus C)$

$p + q \in (A \oplus B) \cup (A \oplus C) \rightarrow p + q \in (A \oplus B) \vee p + q \in (A \oplus C)$

a. $p + q \in (A \oplus B) \rightarrow p \in A \wedge q \in B \rightarrow q \in (B \cup C) \rightarrow p + q \in A \oplus (B \cup C)$

b. $p + q \in (A \oplus C) \rightarrow p \in A \wedge q \in C \rightarrow q \in (B \cup C) \rightarrow p + q \in A \oplus (B \cup C)$

To summarize: $\mathbf{p + q \in p + q \in A \oplus (B \cup C) \rightarrow p + q \in (A \oplus (B \cup C) \rightarrow A \oplus (B \cup C) \supseteq (A \oplus B) \cup (A \oplus C)}$

From (Direction 1) + (Direction 2):

$\mathbf{A} \oplus (\mathbf{B} \cup \mathbf{C}) = (\mathbf{A} \oplus \mathbf{B}) \cup (\mathbf{A} \oplus \mathbf{C})$

2.2 What is the Minkowski sum of:

2.2.1 Two points

A point. Let's set $p_1 = (x_1, y_1),\ p_2 = (x_2, y_2)$. Then the Minkowski sum of p_1 and p_2 is a point with the coordinates $(x, y) = (x_1 + x_2, y_1 + y_2)$

2.2.2 A point and a line

A translated line. The vector equation for a line is:

$$\mathbf{l} = \mathbf{l_0} + t \cdot \mathbf{v},\ \ t \in R$$

where t is the only degree of freedom of the equation.

This can be considered a set of points. Without loss of generality, we consider a 2D line:

$$l = (l_{x,0}, l_{y,0}) + t \cdot (a, b)$$

Let $\mathbf{p} = (x_0, y_0)$ be a point. Then the Minkowski sum of $\mathbf{p}$ and $\mathbf{l}$ is:

$$\mathbf{l} = \mathbf{l_0} + \mathbf{p} + t \cdot \mathbf{v} = (l_{x,0} + x_0, l_{y,0} + y_0) + t \cdot \mathbf{v}$$

This is the line $\mathbf{l}$, translated in space by (x_0, y_0).

2.2.3 Two lines segments (think of all possible cases)

An area(Polygon) or one line. in case the line are co-linear then the geometric place of all the dots should be in a line. in case the two lines are parallel then the geometric place of all the points is one lines. in the remain options for the lines: an area.

2.2.4 Two Disks

A displaced disk with the radius of the sum of the two disks' radii.

2.3 Prove $E \leq 3V - 6$ (assume that $V \geq 3$)

We will prove this inequality using induction.

Base case:

Given $V \geq 3$, we start by analysing a triangle - which is the strongest test, compared to an open chain of 2 edges, or a polygon of order $n > 3$, which leave out edges which could be added without the need to add extra vertices.

$$V = 3, E = 3$$

$$3 \overset{?}{\leq} 3 \cdot 3 - 6 = 3 \rightarrow True$$

Induction Step:

We build upon the base case (a triangle), by adding more triangles. There are 2 cases:

Case 1: This adds 2 edges, and one vertex to the planar graph.

Case 2: This adds 1 edge, and 0 vertices (for example, adding the 4th triangle to create a trapezoid).

We can build up the graph in this fashion to ensure the maximal number of edges to vertices ratio.

Connecting other polygons of order $n > 3$ would always leave out at least one edge that could be added. For example, adding a square leaves out a possible edge - meaning the inequality will be easier to satisfy. Another example is adding a triangle, but connecting it to a vertex of the graph, instead of an edge - This leaves out 2 edges that could be added to create a trapezoidal boundary for no extra vertices.

Thus, to ensure the 'worst case', we only add triangles to our planar graph, which ensures maximal edges.

Assume therefore, that we have a planar graph composed of F triangles connected in the way detailed above. Since each face is made up of exactly 3 edges, and each edge is a boundary of exactly 2 faces, we can say:

$$3F = 2E$$

now recall Euler's equation:

$$V - E + F = 2$$

$$\rightarrow V - E + \frac{2}{3}E = 2 \rightarrow V = \frac{1}{3}E + 2$$

Note that this is the case where the number of edges in the planar graph is maximal, however it is possible to have a graph with fewer edges. Therefore we get the inequality:

$$V \geq \frac{1}{3}E + 2$$

$$\rightarrow \mathbf{E \leq 3V - 6}$$

3 Exact Motion Planning for a Diamond-Shaped Robot

3.1 Code Overview

We are given an environment which consists of simple convex obstacles and a convex robot. Given a starting point and a goal point, we are expected to generate the shortest Euclidian path to the goal.

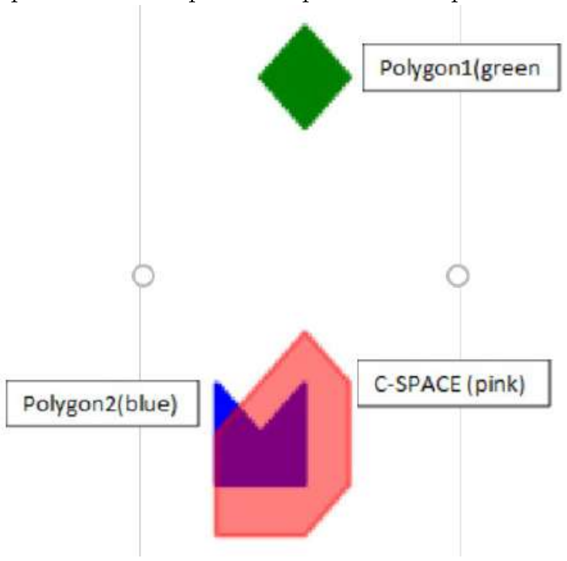
3.2 Constructing the C-Space

Our implementation assume that all the obstacles are convex polygon. that's why we can compute the C-Space of each obstacle in O(m+n) when n is the number of vertices in the obstacle and m - is the number of vertices in our robot polygon (also convex). If the obstacle is non-convex then the complexity that need to solve the problem is by far more complicated. We will see in the following figure that the C-Space even crosses the obstacle, which defeats the purpose of the whole algorithm.

In the example we have the robot: POLYGON ((0 -0.5, 0.5 0, 0 0.5, -0.5 0, 0 -0.5)) and obstacle: POLYGON ((0 0, 1 0, 1 1, 0.5 0.5, 0 1, 0 0))

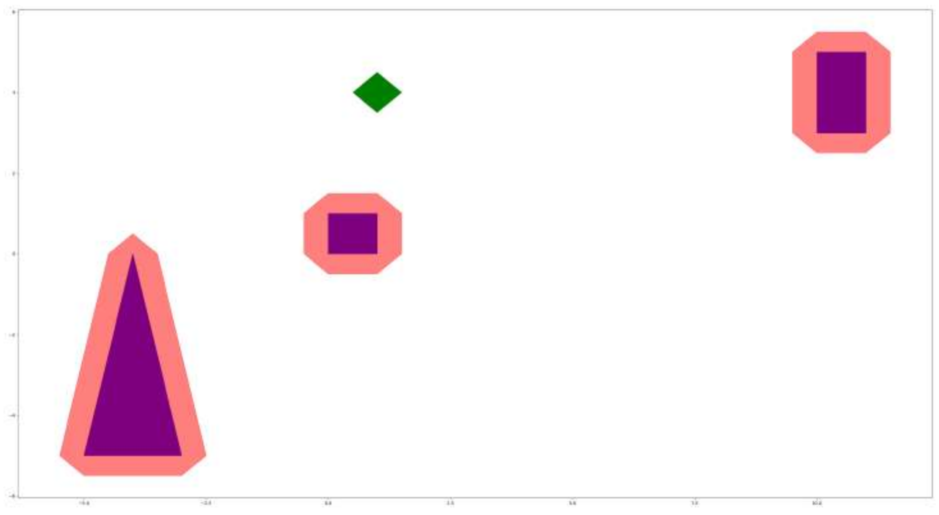
After we add the vertex (1,1.5) to the C-Space and continue CCW the cross between the current vector on the obstacle (from (0.5,0.5) to (0,1)) and the current vector on the robot polygon(from (-0.5,0) to (0,-0.5)) is equal to zero -¿ hence we rotate both by +1 and finally add the point (0,0.5) to the C-Space which make the problem of crossing the obstacle in the C-Space (the edge added between (1,1.5) to (0,0.5) cross the obstacle edge((0.5,0.5) to (0,1))).

Figure 1: Example of an attempt to compute the C-Space with concave polygon



Nevertheless, since we are assured a completely convex workspace, we manage to get the correct result:

Figure 2: C-Space Generated for Convex Polygons

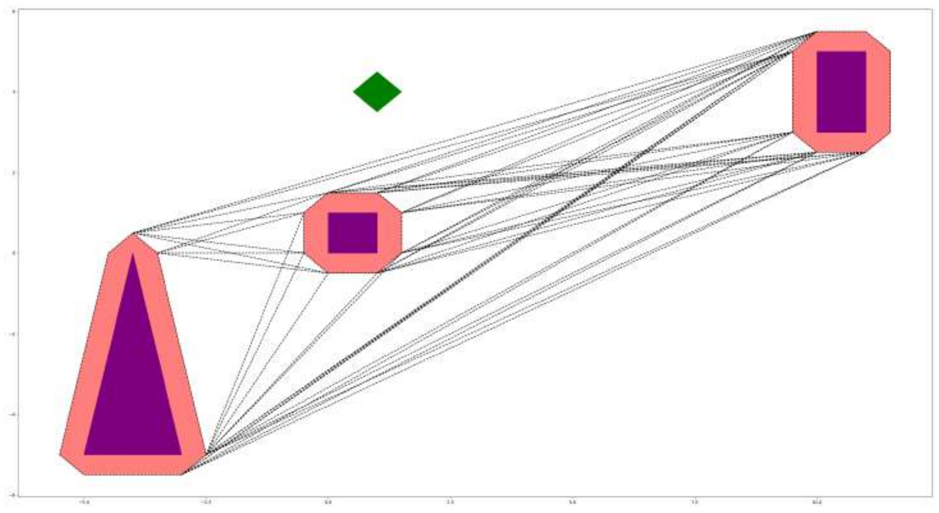


3.3 Building the Visibility Diagram

The visibility Diagram was generated in a Naive way. We create a list of all the vertices from the C-space obstacles , the robot, the start, and the end ($m + n + 2$ total). We list through each vertex ($n + m + 2$ total), and create an edge with all of the other vertices ($n + m + 2$ total). If an edge is in collision, it is ignored (n iterations to check for single collision).

All in all, our algorithm has a time-complexity of $O((n + m) + (n + m)^2 \cdot (n))$. If we assume that the number of vertices of the obstacles is greater than the number of vertices from our robot (4), then we can say the time-complexity is $O(n^3)$. This is only acceptable since we are dealing with simple convex shapes, meaning the value of n is quite low. It is possible to improve the complexity by using better methods, such as scanning and maintaining a balanced search tree.

Figure 3: Visibility Graph generated from the vertices of the C-Space obstacles



3.4 Computing Shortest Paths

For calculating the shortest paths, we make use Dijkstra's algorithm.

3.4.1 Dijkstra

This algorithm will allow us to find the true cost of traversing from the start vertex to any other vertex, while also keeping track of the paths to reach all of these vertices with their minimal costs.

Firstly, we receive a collection of lines from the visibility graph. We then sort all of the edges into an adjacency list ($O(|E|)$ time). Each key represents a vertex, and the values stored are the vertex's neighbours, as well as the cost (Euclidian distance to the neighbour in this case). We also maintain a few lists. A 'cost list', which shows the current shortest cost to a vertex from the starting vertex, and a 'previous list', which tracks a vertex's predecessor in a path which ensures the current shortest path to the start vertex. Finally, we have a priority queue, which orders the unvisited vertices by distance from the start vertex.

We start by assigning infinity to all the distances in the cost list and None to all previous vertices in the previous list ($O(|V|)$ time), and set the start vertex distance in the cost list to 0 (O(1) time).

We then pop the vertex with the smallest cost from the priority queue ($O(|V|)$ time). We create the priority queue ($O(|V|)$ time). While the priority queue is not empty, we remove the vertex closest to the start, and check all its neighbours. If it is possible to improve the neighbour's distance to the start by going through the current vertex, we update its distance value in the cost list. This change is also reflected in the queue (O(1) time).

All in all, Dijkstra takes $O(|V|^2)$. This can be improved by making use of a binary heap for the priority queue, which would improve run-time to $O((E+V)logV)$.

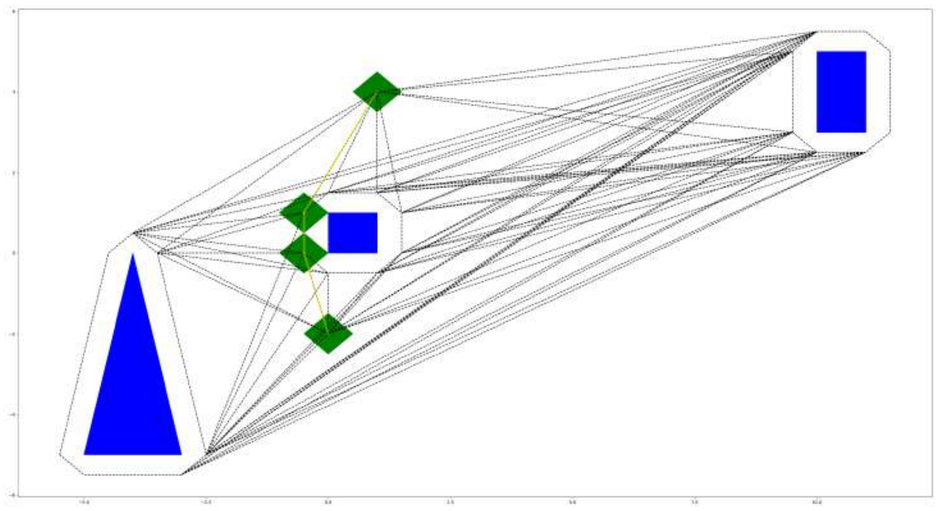
3.4.2 Shortest Path

Once we have the true costs and paths to the vertices, it is very simple to find the cost and path from the start vertex to the end vertex.

First we find the cost to traverse to the end vertex ($O(|V|)$), and then we create an empty list and add the end vertex to the list (O(1)). We now iteratively check the parent vertex of the current vertex, and add it to the list, so long as it is not NONE ($O(|V|)$).
All in all, this step takes us $O(|V|)$ time to complete (time could also be improved slightly using a better data structure).

The result generated is as follows:

Figure 4: Shortest Path Found



This seems correct, as the solution found moves along the boundary of the C-Space obstacles. As learnt in class, the shortest path should be 'wrapped' along the boundaries.

3.5 Final Run

We rerun the algorithm for different obstacles, start position, end end position. We show an extreme example where 2 C-Space convex obstacles touch each other.

Figure 5: C-Space Generated for Convex Polygons

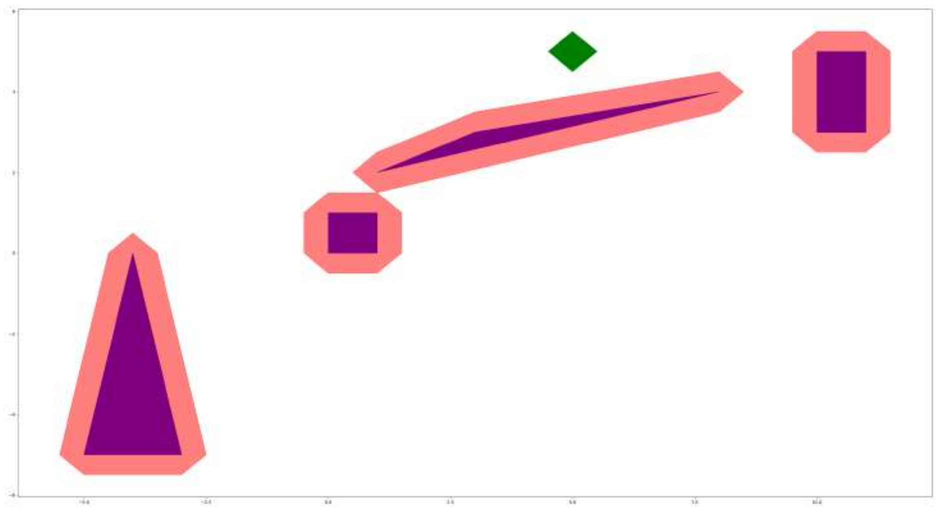


Figure 6: Visibility Graph generated from the vertices of the C-Space obstacles

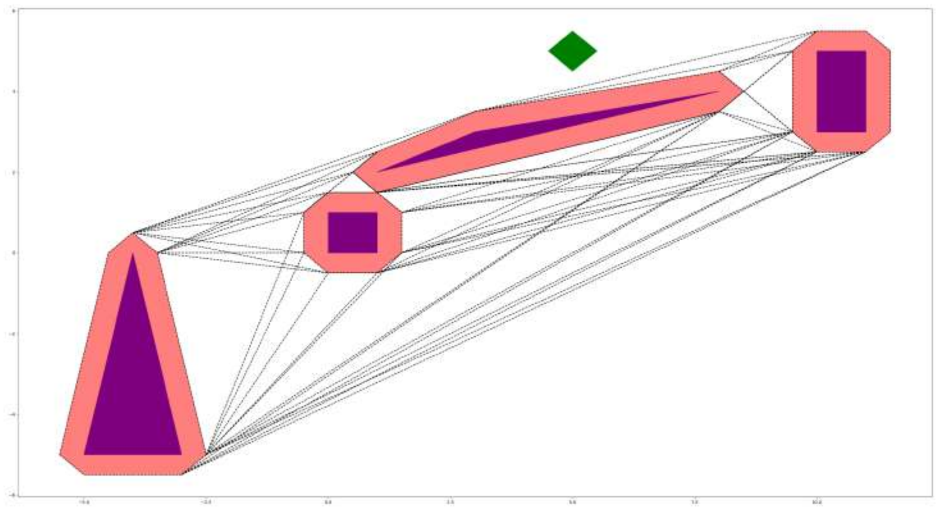


Figure 7: Shortest Path Found