Robot Movement in CGAL: Path Discovery

Sergios Aftsidis*

University of Athens - Department of Informatics and Telecommunications
std09052@di.uoa.gr

Abstract

Moving in a space with obstacles is a problem that might seem trivial for a human being, because of our understanding of space and obstacles. Thus, when we try to program a robot to do so, we are faced with many problems. This report tries describes an approach to discretize the space, mark the obstacles and finally find the optimal path (if one exists) from one point to another. Also an implementation is provided, and the routines - functions - approaches used are described. The project was implemented using the Computational Geometry Algorithms Library CGAL (`http://www.cgal.org/`*.*

Contents

*Supervisor: Ioannis Emiris

I. Introduction

The problem, in further detail, can be described as:

Problem Definition. *Given a **bounding box**, **N obstacles** (polygons), a starting point -point **A** (robot)- and an ending point -point **B** (destination)-, find the **optimal path** (if one exists) from point A to point B.*

Some specifications:

- **bounding box:** The bounding box is described as a polygon (convex or non)
- **obstacles:** The obstacles are also polygons (convex or non) ***Degenerate Cases***
- **robot:** The robot is a point with no dimensions
- **optimal path:** The optimal path is the path using the less number of ***steps***

To provide a solution to such a problem, the key is to represent our environment in a convenient manner. The first issue that comes to mind would be to represent the bounding box as a polygon and then place the obstacles, the starting point and the ending point inside the polygon. This idea, even though comes really naturaly and seems adequate, is not functional. ***But why?*** Imagine the following scenario. We place all these elements in the bounding box. Now it is time to start our path from point **A** to point **B**. How are we going to move in a continuous space? This is a big "gap" in our representation, because movement now seems really abstract and disoriented. It is obvious that we somehow need to provide some **discrete options** for every movement. This approach is called ***Discretization***, and can be achieved using various algorithms. In our case, we have selected to use ***Constrained Delaunay Triangulation***

II. Discretization

In the section above we mentioned that we are going to use a **Constrained Delaunay Triangulation** to create a discrete space. Below we will provide some advantages this approach has.

I. Triangulation

A **triangulation** (among others) provides a way of separating an object into smaller, triangular regions. We specify some points inside the object, and after the triangulation we have triangles which form the actual object [1] . It's general time complexity is $\Theta(nlogn)$ which provides us with a relatively fast way to separate our object in discrete regions.

II. Delaunay Triangulation

A **Delaunay Triangulation**, is a triangulation with a special property:

Definition. *A **Delaunay triangulation** for a set P of points in a plane is a triangulation **DT(P)** such that no point in P is inside the circumcircle of any triangle in **DT(P)**. Delaunay triangulations maximize the minimum angle of all the angles of the triangles in the triangulation; **they tend to avoid skinny triangles**.*

This property is really important in applications such as ours. Imagine the following case:

(a) *Points A, B lie in the same triangle in a non-Delaunay Triangulation*

(b) *A Delaunay Triangulation guarantees that the space will be partitioned more "balanced", and such cases are probably avoided*

Figure 1: *Delaunay Triangulation importance*

In this case, points A and B are far away from each other, but a random triangulation has placed them in the same triangle, and by extend in the same region, meaning they are connected. As we can see, it is essential to have a balanced discretization of our space, and the Delaunay Triangulation guarantees that (in most cases).

III. Constrained Delaunay Triangulation

A Constrained Delaunay Triangulation is a constrained triangulation that "tries" to be Delaunay. A constrained triangulation is a triangulation where some edges are forced (apart from the specified vertices,which are also forced). This is the key element in our representation. Both the bounding box and the obstacles are inserted as constraints in the triangulation, and we triangulate between them, as illustrated below:

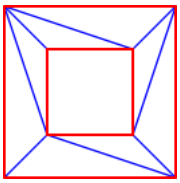


Figure 2: *Triangulating between polygons: The outer polygon represents the bounding box, whereas the inner represents an obstacle*

III. Path Discovery

After we have discretized the plane, path planning is implemented trivially. In this implementation, the plane is treated as a connected graph, and thus the **BFS** search algorithm is used. It finds the **shortest path** from point A to point B in time linear in the size of the graph. The reason that suggests the use of this approach is the way $CGAL$ implements the Constrained Delaunay Triangulation (see **CGAL::CDT**)

IV. Implementation

This section describes some implementation-specific issues. Among others, a short description of the $CGAL$ classes and functions used is provided. For a more complete documentation, one is

referred to the *CGAL* :: *2D_Triangulation* **manual**

I. Basics

II. CGAL::Constrained_Delaunay_Triangulation_2

II.1 General

CGAL implements a Triangulation in the following manner:

Definition. *A 2D triangulation in CGAL can be viewed as a* ***planar partition*** *whose bounded faces are triangular and cover the convex hull of the set of vertices. The* ***single unbounded face*** *of this partition is the complementary of the convex hull. A fictitious vertex, called the* ***infinite vertex*** *is added to the triangulation as well as* ***infinite edges*** *and* ***infinite faces*** *incident to it. Each infinite edge is incident to the infinite vertex and to a vertex of the convex hull. Each infinite face is incident to the infinite vertex and to a convex hull edge.*

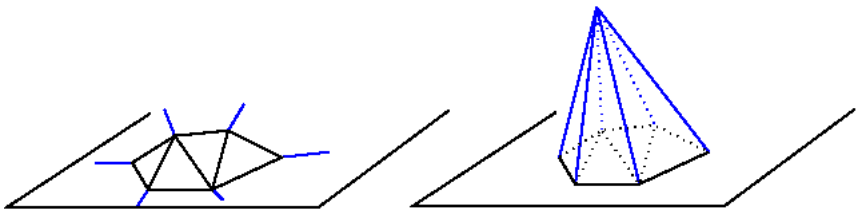


Figure 3: *Finite* ***black*** *and Infinite* ***blue*** *edges*

Because a triangulation is a set of triangular faces with constant-size complexity, triangulations are not implemented as a layer on top of a planar map. CGAL uses a proper internal representation of triangulations based on faces (in our case **triangular faces** or **triangles**) and vertices rather than on edges. Such a representation saves storage space and results in faster algorithms .

The basic elements of the representation are vertices and faces. Each triangular face gives access to its three incident vertices and to its three adjacent faces. Each vertex gives access to one of its incident faces and through that face to the circular list of its incident faces (**Figure 4**).

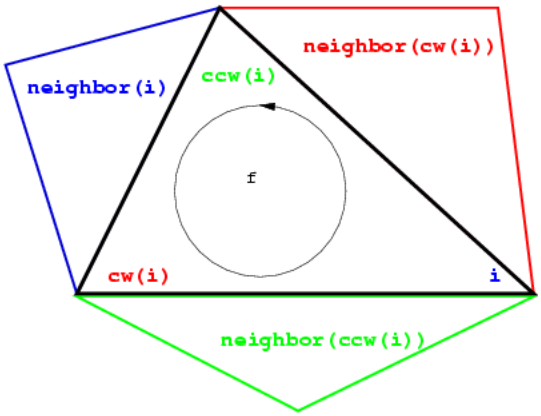


Figure 4: *The three vertices of a face are indexed with 0, 1 and 2 in counterclockwise order. The neighbors of a face are also indexed with 0,1,2 in such a way that the neighbor indexed by i is opposite to the vertex with the same index.*

II.2 Adding Objects

As previously mentioned, both the **bounding box** and the **obstacles** are added as constrained edges on the triangulation. Thus, the process of adding obstacles is no other than the process of adding their edges. For each edge added, the triangulation is altered to mantain its properties (triangular faces, Delaunay property). One may wonder, is the resulting triangulation **always** Delaunay? The answer is no, but it is "**Delaunay Adjacent**".

So far we have entered both the bounding box and the obstacles as constraints in a Constrained Delaunay Triangulation. The immediate question rising is, how are we able to distinquish between them? The following section describes this process.

II.3 Marking the obstacles

From our current representation there is **no** obvious way to distinquish between the obstacles and the bounding box. One might think that a polygon object as a **hole**,but this is **not** the case in our representation. The **obstacles are triangulated too**. So we need a way to separate the **faces that belong in the obstacles** and the **faces of our triangulated plane** (the ones we are free to move on).

The following procedure starts from the **infinite face**, and continues onwards, marking each domain with a **nesting level**.

Explore set of facets connected with non constrained edges,and attribute to each such set a nesting level. Start from facets incident to the infinite vertex, with a nesting level of 0. Then, recursively consider the non-explored facets incident to constrained edges bounding the former set and increase the nesting level by 1. Facets in the domain are those with an odd nesting level.

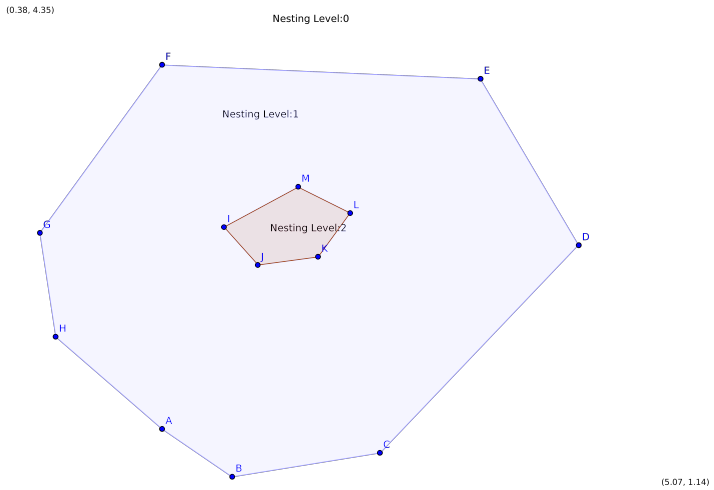


Figure 5: *The Infinite face has nesting level 0, the next domain (the bounding box) 1, and the obstacles 2*

This requires an extra variable for each facet, to store the nesting level. *CGAL* provides us with a way to "add" extra information on every triangulation face. This is implemented with the **struct Faceinfo** structure, that we override to store any information we require. In our case, we included the following:

```
struct FaceInfo2
{

  int nesting_level;
  bool visited;
  bool in_domain(){
    return nesting_level%2 == 1;
  }
  FaceInfo2(){
        is_constr=false;
  }
};
```

II.4 Locating the start point

The *CGAL* :: *Constrained_Delaunay_Triangulation_2* provides us with functions that locate given points on a triangulation facet. From the manual:

The triangulation class provides a method to locate a given point with respect to a triangulation. In particular, this method reports whether the point coincides with a vertex of the triangulation, lies on an edge, in a face or outside of the convex hull.

In detail, we use the *locate*(*Point_2*&) function. This returns the face of the triangulation where the point lies.

II.5 Searching for the shortest path

Since the facet in which the point lies has been determined , searching for the shortest path is a rather simple procedure. We follow the *BFS* algorithm, moving along each face's neighbors, to find the shortest path first. One **important modification** has been implemented. When the algorithm checks a neighbor, it discards the facets which are outside the current domain, because **these are obstacles**. Thus, the algorithm replies with the shortest path, if it exists.

V. Known Issues

I. Obstacles Intersecting Bounding Box

Due to the way the domains are marked, a **limitation** occurs. If the edge of some obstacle lies on a constrained edge, **this obstacle will be marked as free domain**, and path planning will not be able to avoid it. This does not affect obstacles that have **a vertex on a constrained edge**, or even **obstacles larger than the bounding box**. So the only degenerate case / limitation is that all the obstacles' edges must not lie on a constrained edge.

I.1 Suggestion

One possible way to address this issue would be to add some **"noise"** in the obstacle points, to change the edges' gradient. This way the edge might intersect the bounding box but it will definitely not lie exactly on one of it's edges. This introduces problems though. If *e.g.* we add noise and a vertex that used to be on a constrained edge lands on the interior of the bounding box,

a possible path might be generated where there was none. If the noise is small enough the whole picture will not be altered enough, but we cannot say with certainty what values would be safe.

II. Triangulation Issues

Another issue rises when the bounding box is non-convex. From the 2D_Triangulation manual:

The triangulations of CGAL are complete triangulations which means that their domain is the convex hull of their vertices. Because any planar triangulation can be completed, this is not a real restriction. For instance, a triangulation of a polygonal region can be constructed and represented as a subset of a constrained triangulation in which the region boundary edges have been input as constrained edges.

This means that if we input constraints that form a non-convex polygon, the triangulation **will include extra facets** outside this polygon, to form the **convex hull** of the constraints specified. This **does not affect** our implementation, though, as these facets will be assigned nesting level 0, as are the infinite faces.

VI. A complete Example

Here a complete execution example is described (input, output, visualization of results).
Please note that for the program to function properly, the directory it is executed must include a folder named *"data"*, and the subfolders *"data/input"* ,*"data/input/obstacles"* , *"data/output"* .

I. Input files

The program takes as input files from the "./data/input" directory, in .csv format,as such:

file.csv

point1.x (tab) point1.y
point2.x (tab) point2.y
etc..

The files that the program reads as input are described below

- **bounding box:** Directory: *"./data/input"* Name: *b_box.csv*
- **obstacles:** Directory: *"./data/input/obstacles"* Name: *(anything)*

Upon execution the program loads the bounding box and the obstacles, and prompts the user to enter the 2 points (*start-end*). The points **must** be inside the bounding box.

II. Output files

In a similar way, the program outputs .csv files that contain the points of the triangulation.The output is provided in 3 lines per triangle (3 points) separated by empty lines. The following attributes are provided (directory:"./data/output"):

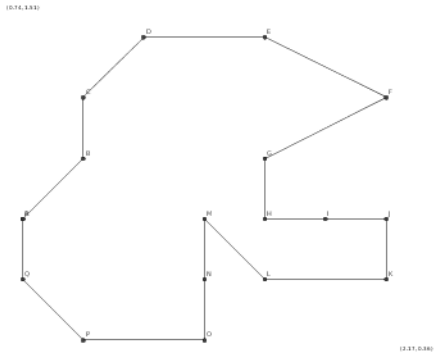
- **triangulation:** Name: *triangulation_extra.csv* **Description:** This file contains the whole convex hull of the triangulated area (Info)

- **free_domain:** Name: *triangulation.csv* **Description:** This file contains the domain marked as free (the domain the robot is free to move in)
  Also the input bounding box and obstacles are provided (in the files named respectively).

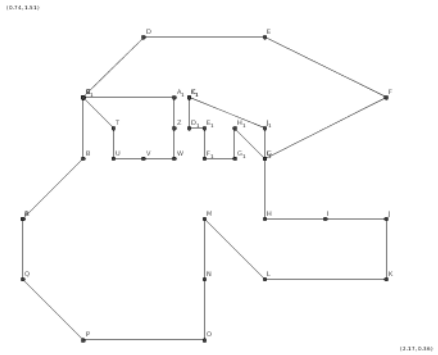
III. Execution - Images

III.1 A general example with an available path

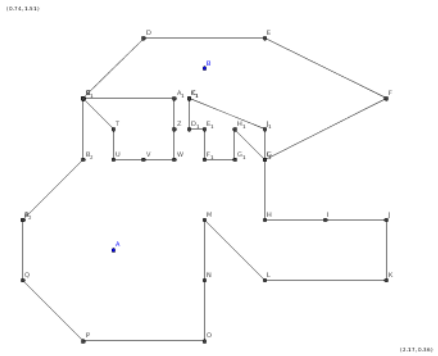
Imagine the following bounding box



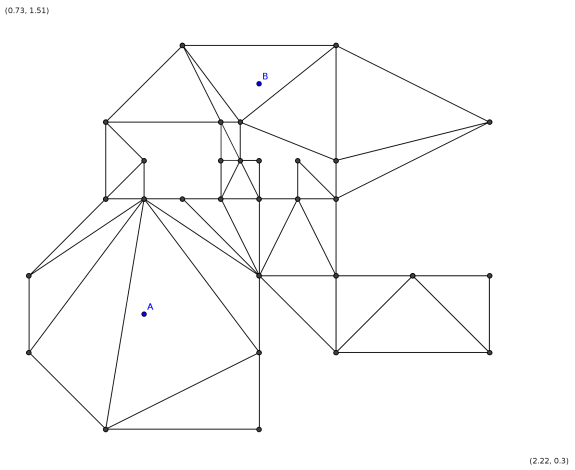
And now imagine the following obstacles



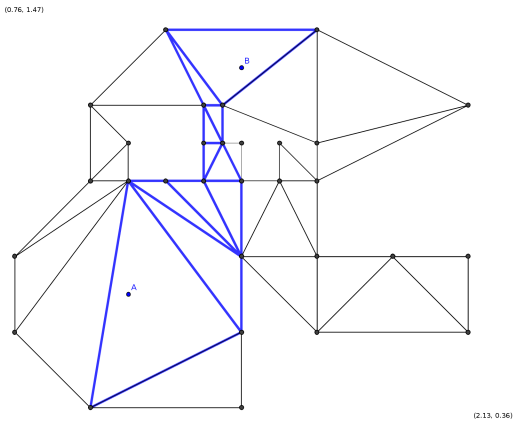
Say we want to move from point A to point B, for the points below



From the output files we can see the triangulation and from the execution output we can see the path chosen.



(a) *The bounding box with the triangulation of the free domain*



(b) *The optimal path found*

The execution output is:

```
The optimal path was found
Printing the coordinates of the triangles that form it
1.2 1.4;1.35 1.2;1.6 1.4
1.3 1.2;1.35 1.2;1.2 1.4
1.35 1.1;1.35 1.2;1.3 1.2
1.3 1.2;1.3 1.1;1.35 1.1
1.3 1.1;1.3 1;1.35 1.1
1.3 1;1.4 1;1.35 1.1
1.4 0.8;1.4 1;1.3 1
1.4 0.8;1.3 1;1.2 1
1.4 0.8;1.2 1;1.1 1
1.1 1;1.4 0.6;1.4 0.8
1 0.4;1.4 0.6;1.1 1
The path is printed.. (Order: END -> START)
```

The input / output files of this example are available in the file *exp_data.zip* Also, in the directory *"./data/input"* the file *polygon_4.csv* is included. If this file is placed in the *"./data/input/obstacles"* directory, it blocks all paths for the points described above. It can be used to test an example where no path is found.

References

[Lozano-Perez , Wesley, 1979] An algorithm for planning collision-free paths among polyhedral obstacles

[Chaitanya A. Deosthale 2012] Path Planning in Planar Environments using Triangulation

[Mariette Yvinec] CGAL 4.4: 2D Triangulation : User Manual