



UpWind Project

January-May, 2006

Project Supervisor: Samuli Saukkonen

Project Members:

★ *Victor Arroyo Valle*

★ *Jose Cano Mendoza*

★ *Vicente Villamón Ribés*

*Department of Information Processing Science
University of Oulu, Finland*

A familia y a mis amigos por creer en mí.

Victor Arrollo Valle



*A toda la gente que me ha apoyado
durante este tiempo porque ellos
han hecho posible que mi sueño se haga realidad.
Muy especialmente a Bienvenido, Rosa M^a, Sara y Giovanna.*

Jose Cano Mendoza



*A mis padres por apoyarme,
a mi hermana por animarme, a tí y
a mis dos compañeros por soportarme.*

Vicente Villamón Ribés



*To Samuli to let us work so freely.
Freedom is the key to achieve the success.*

UpWind Members



Abstract

Navigation software is not a simple. The main goal is to get the optimal path between two or more points located in a navigation map using the maximum ship speed. Between this two points may exist factors that can influence in the final trajectory of the sail, so scan all these factors its needed to trace the optimal route. Those factors are islands, sets of rocks, low depth areas, other boats, wind speed and direction and obligatory navigation routes. Is not possible to analyze and solve the problem with all this kind of different factors, so we must divide the whole issue in smaller subproblems that can be handled easily.

We need to develop an algorithm to avoid those factors and get the optimal path. To get it, we are going to develop a simulator; in principle, we will not work with real information because this information is not usable for us so we can handle information more suitable for us.

Firstly, we will take the most simple problem, we will work with the easiest object which is the biggest object at the same time: islands. We see them like geometrical objects because it is needed to formulate the real problem like a mathematical issue in order to get the appropriate algorithm.

When the first problem is solved we will introduce the other factors one by one, to finally see the final solution for the whole problem.

Contents

1	Motivation	1
2	First Algorithm	5
2.1	Tangents: Point to Polygon	6
3	Shortest path problem	9
3.1	Overview Dijkstra's Algorithm	11
3.1.1	Directed Graphs	11
3.1.2	Greedy Algorithms	12
3.1.3	Dijkstra's Algorithm description	13
3.1.4	Dijkstra: Example 1	15
3.1.5	Dijkstra: Example 2	19
3.1.6	Related problems, applications and algorithms	25
4	UpWind Simulator	27
4.1	Input: Map file format	29
4.2	Upwind Simulator: User Interface	30
4.2.1	Main form	30
4.2.2	File Menu	31
4.2.3	System Menu	32
4.3	Processing info & algorithm implementation	33
4.3.1	Objects definition	33
4.3.2	Whole process overview	34
4.3.3	Data parsing	35
4.3.4	Drawing map info	37
4.3.5	Drawing and storing control points	38
4.3.6	Handling information and performing calculations	38

4.3.7	Result: Optimal path	55
5	Project final evaluation	57
5.1	Achievements	58
5.2	Next steps	59
6	Appendix	63
6.1	UpWind Simulator class overview	64
6.1.1	Simulator.java	64
6.1.2	Map.java	64
6.1.3	CVisibilityGraph.java	65
6.1.4	Dijkstra.java	65
6.1.5	Path.java	65
6.1.6	CPolygon.java / CVertex.java / CEdge.java	65
6.2	Resources used	66
6.3	Project Logo	67
6.4	Acronyms	68

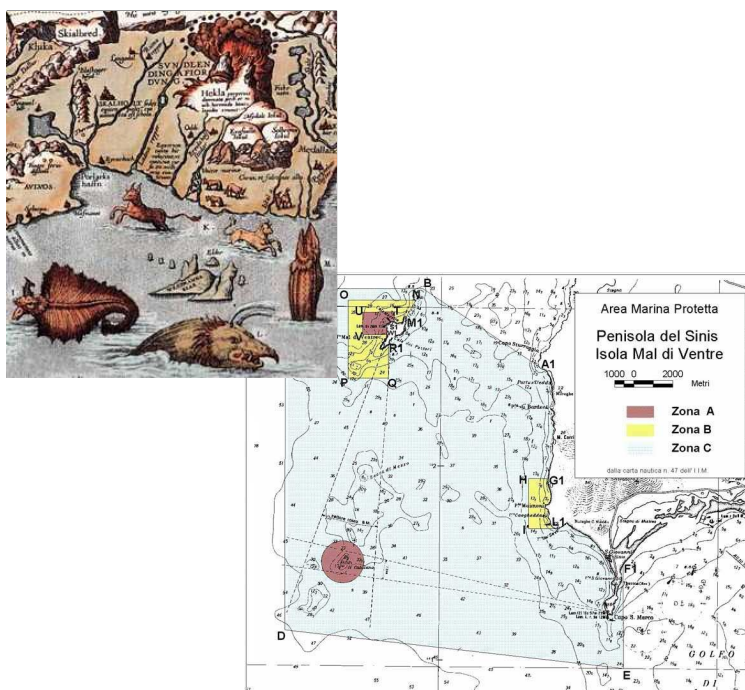
Motivation

“Sailing is not easy. There are lots of factors that you should bear in mind if you want to navigate fast and safe: weather forecast, boat characteristics, course calculation... Those factors are highly variable and individual for each ship; plus the psychical handling of the boat could make navigation hard and unpleasant for a single skipper ”.

With this sentence we started our project, it was January 2006. We thought about the problem from a practical point of view. Sailing is a very nice activity, but it can be also very dangerous; the variety of users and configurations is infinite.

Based on personal needs, we researched the available navigation software in the market looking for a suitable product that fulfilled our customer needs. The results of our research were shocking: the most of the analyzed products were over-featured, too basic or just too expensive. Nowadays terrestrial GPS systems are widely used, they are cheap, reliable and easy to use. There are many companies developing hardware and software, and we can see a growing market around the area. GPS comes as an option in the most of car models. The devices integrate multimedia content, comfort controls and the advanced models can interact with cellphones via Bluetooth connection. Those GPS devices bring the drivers real-time traffic information system, road status and route planning. Our motivation is to create the software that allows to have all this functionality while navigating freely in the sea.

Our research started by analyzing the evolution of Marine GPS systems. As we see in the illustrations, the evolution is more than obvious: from inexact maps, we can get sub-meter high resolution satellite pictures from anywhere in the world, or high detailed sea maps with depth-profiles and ocean streams. Internet has brought real time weather forecast information to the ships, and captains have usually a high-level education in order to get the required license to sail.



Some other factors have evolved as well: GPS precision has been increased since 1957, year when the system was used for the first time, till today; going from periodic hour-transmissions from five satellites to a real time information provided by a vast network of satellites offering a four meter resolution for non-military purposes. The sail boats have gained reliability and technology, they are much safer now and they can be driven by a single sailor with the help of very advanced auto-pilot systems.

The possibility of adapting the existing terrestrial GPS navigation systems was discarded: they do not take in consideration external factors such as weather conditions or status of the road and because no special techniques are required in order to drive a car, the adaptation of those systems to the purpose of our project was impossible.

After our previous research we all agreed on making our own software, 100% committed with the purpose of our project; we called it the UpWind Project.



We settled our own objectives based in our own requirements and stroked forward in order to achieve them. Our basic project plan included tasks such as gathering all the available information around the boat (weather forecast, sea maps and polar diagrams) and incorporate it to an OpenSource, richly featured solution that covers the basic sailing needs: route planning and distance calculation.

We will help captains by providing real-time and reliable information in order to make the navigation safe and easy. We will also help disabled sailors and non-experienced captains to make every single cruise a pleasure for both passengers and crew. Among the basic needs our software can be crucial in emergency situations such as low-visibility navigation conditions. This our main motivation.

The main objective of the UpWind Project is to create intelligent algorithms for route planning and optimization. Those mathematical algorithms will become the base of a highly reliable piece of OpenSource Software that the whole community will be able to use and modify in order to full fill every individual need.

2

First Algorithm

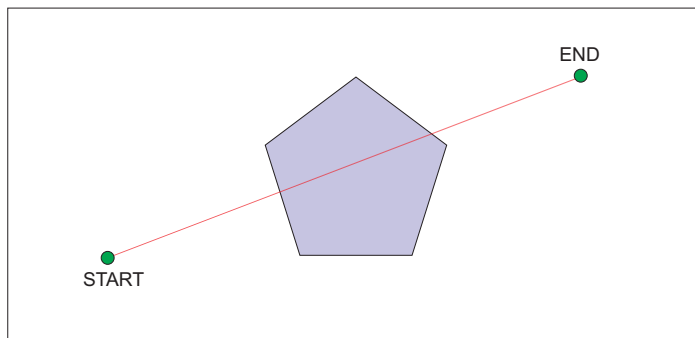
Contents

2.1	Tangents: Point to Polygon	6
-----	--------------------------------------	---

We started to work with the simplest problem, avoiding islands.

Our first idea was to work with one island (polygon) and two points. The first one, would be the point where the ship was and the other point would be the point where the ship wanted to go.

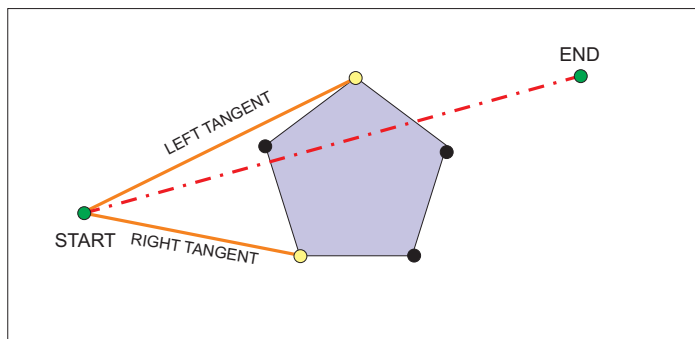
We needed to transform the real problem in a geometrical problem so we could do it working with three figures, one polygon (island) and two dots (start and end point). Then, tracing the segment between the dots we realized that the polygon was in the middle so in this moment, we defined the problem.



After hours of research ¹, one of the first proposals to tackle the situation could be try to find the tangents to the polygon from the start point.

2.1 Tangents: Point to Polygon

Based on resources found in some sites, it was possible to calculate tangents between one point and one polygon. After analyzing the below figure we could appreciate that between one point and one polygon would exist always two tangents called left and right tangent.



¹Accessed 30/01/06: http://softsurfer.com/algorithm_archive.htm

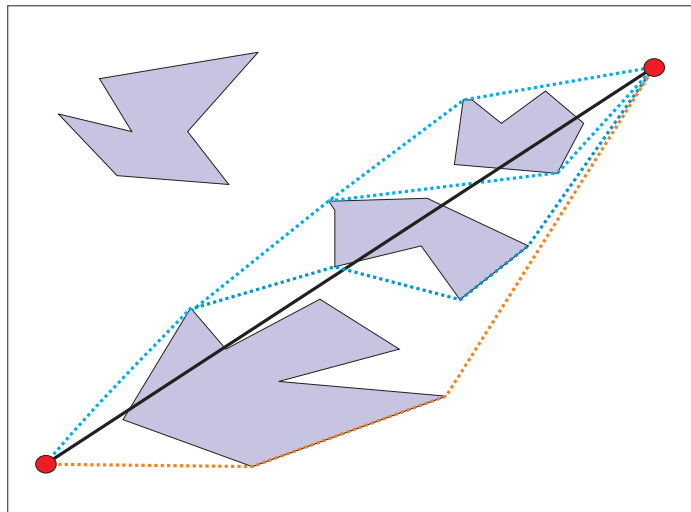
FORCE BRUTE POINT TO POLYGON ALGORITHM

```

1
2 input: L = {V0,V1,...,Vn-1,Vn=V0} is any 2D polygon
3       P = a 2D point
4
5       VR = VL = V0;
6       for each vertex Vi (i=1,n-1)
7         {
8           eprev = (P is left of Vi-1Vi); // left of edge ei-1
9           enext = (P is left of ViVi+1); // left of edge ei
10          if ((NOT eprev) and enext) {
11            if (Vi is not below VR) // handles nonconvex case
12              VR = Vi;
13          }
14          else if (eprev and (NOT enext)) {
15            if (Vi is not above VL) // handles nonconvex case
16              VL = Vi;
17          }
18        }
19
20 return tangent points VR and VL

```

With this algorithm was possible to avoid our simple problem but we had to step and try to get a more complex problem. After analyzing the same situation but with more factors.



It is really easy to see in the drawing that our algorithm could be fine for this problem because we could arrive to the end point easily calculating tangents and recalculating them each time we were in a new vertex.

In fact the following pseudocode algorithm could be fine to reflect the solution of the last drawing:

```

1  FIND ROUTE - PSEUDOCODE ALGORITHM
2  FindRoute:
3
4      - Trace rule between 2 points [start ,end]
5      - Calculate intersection between the rule and polygon
6      if intersection exists{
7          - calculate tangents
8          - start point = tangent point
9          - Go To FindRoute
10     }
11     else{
12         - Trace rule
13         return Path_Calculated
14     }

```

But, It was not so easy. We had to think in the following factors:

- Would it be possible to know which would be the polygons which were in our route?
- Would it be easy to avoid useless polygons?
- Would it be flexible to handle dynamic factors in the future with this algorithm? [wind speed, boats, rocks] Would it be easy?
- Should we have trusted in something never tried?

Before implementing all these ideas we decided to go on with our own research and try to find out if there was another solution to this kind of problem. We were sure that there were other kind of algorithms used in robotics that were better than our own solution and that they have been already tried for thousands of companies and projects.

Shortest path problem

Contents

3.1	Overview Dijkstra's Algorithm	11
3.1.1	Directed Graphs	11
3.1.2	Greedy Algorithms	12
3.1.3	Dijkstra's Algorithm description	13
3.1.4	Dijkstra: Example 1	15
3.1.5	Dijkstra: Example 2	19
3.1.6	Related problems, applications and algorithms	25

Our goal was getting the optimal route between two or more given points. This optimal route had not be calculated referring only to the distance. We would have some other factors altering the optimal route like wind speed, wind direction, etc... and we would have to take them into account.

Shortest path problem had already been thought, solved and used in hundreds of real applications like in robotics.

In graph theory, the single-source shortest path problem is the problem of finding a path between two vertexes such that the sum of the weights of its constituent edges is minimized. More formally, given a weighted graph (that is, a set V of vertexes, a set E of edges, and a real-valued weight function $f : E \rightarrow \mathbb{R}$), and given further one element n of N , find a path P from n to each n' of N so that

$$\sum_{p \in P} f(p)$$

is minimal among all paths connecting n to n' . The all-pairs shortest path problem is a similar problem, in which we have to find such paths for every two vertexes n to n' .

A solution to the shortest path problem is sometimes called a pathing algorithm. The most important algorithms for solving this problem are:

- ***Dijkstra's algorithm*** : solves single source problem if all edge weights are greater than or equal to zero. Without worsening the run time, this algorithm can in fact compute the shortest paths from a given start point s to all other nodes.
- ***Bellman-Ford algorithm*** : solves single source problem if edge weights may be negative.
- ***A* algorithm (or A* pathing algorithm)*** : a heuristic for single source shortest paths.
- ***Floyd-Warshall algorithm*** : solves all pairs of shortest paths.
- ***Johnson's algorithm*** : solves all pairs shortest paths, may be faster than Floyd-Warshall on sparse graphs.
- ***Perturbation theory*** : finds (at worst) the locally shortest path

3.1 Overview Dijkstra's Algorithm

Dijkstra's algorithm, named after its discoverer, Dutch computer scientist Edsger Dijkstra, is an algorithm that solves the single-source shortest path problem for a directed graph with non-negative edge weights.

The main idea in this algorithm consists of, being exploring all shorter ways than leave from the origin vertex and arrive to all the other vertex; when the shortest way is obtained from the vertex origin, the algorithm stops. The algorithm relies on the well-known strategy like Greedy algorithms

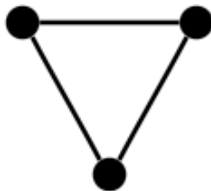
Before further analysis of the algorithm, it's really important to focus in the comprehension of this two concepts: greedy algorithms and directed graphs.

3.1.1 Directed Graphs

Graphs

In mathematics and computer science a graph is the basic object of study in graph theory. Informally speaking, a graph is a set of objects called points or vertexes connected by links called lines or edges. In a graph proper, which is by default undirected, a line from point A to point B is considered to be the same thing as a line from point B to point A. In a digraph, short for directed graph, the two directions are counted as being distinct arcs or directed edges. Typically, a graph is depicted in diagrammatic form as a set of dots (for the points, vertexes, or nodes), joined by curves (for the lines or edges).

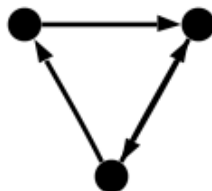
Undirected Graphs



A graph or undirected graph G is an ordered pair $G := (V, E)$ that is subject to the following conditions:

- V is a set of vertexes or nodes.
- E is a set of unordered pairs of distinct vertexes, called edges or lines.
- The vertexes belonging to an edge are called the ends, endpoints, or end vertexes of the edge.

Directed Graphs



A directed graph or digraph G is an ordered pair $G := (V, A)$ with:

- V , a set of vertexes or nodes.
- A , a set of ordered pairs of vertexes, called directed edges, arcs, or arrows. An edge $e = (x, y)$ is considered to be directed from x to y ; y is called the head and x is called the tail of the edge.

In a directed graph, vertexes have both "indegrees" and "outdegrees": the indegree of a vertex is the number of arcs leading to that vertex, and the outdegree of a vertex is the number of arcs leading away from that vertex.

A vertex with an indegree of 0 is called a source (since one can only leave it) and a vertex with an outdegree of 0 is called a sink (since one cannot leave it). It is relatively easy to see that a directed graph with no cycles has at least one source and one sink.

3.1.2 Greedy Algorithms

Greedy Algorithms are algorithms which follow the problem solving meta-heuristic of making the locally optimum choice at each stage with the hope of finding the global optimum. For instance, applying the greedy strategy to the traveling salesman problem produces the following algorithm: "At each stage visit the unvisited city nearest to the current city".

In general, greedy algorithms have five pillars:

1. A candidate set, from which a solution is created.

2. A selection function, which chooses the best candidate to be added to the solution
3. A feasibility function, that is used to determine if a candidate can be used to contribute to a solution
4. An objective function, which assigns a value to a solution, or a partial solution, and
5. A solution function, which will indicate when we have discovered a complete solution

There are two ingredients that are exhibited by most problems that lend themselves to a greedy strategy:

Greedy Choice Property: We can make whatever choice seems best at the moment and then solve the subproblems arising after the choice is made. The choice made by a greedy algorithm may depend on choices so far. But, it cannot depend on any future choices or all the solutions to the subproblem, it progresses in a fashion making one greedy choice after another iteratively reducing each given problem into a smaller one. This is the main difference between it and dynamic programming. Dynamic programming is exhaustive and is guaranteed to find the solution. After every algorithmic stage, dynamic programming makes decisions based on the all the decisions made in the previous stage, and may reconsider the previous stage's algorithmic path to solution. Greedy algorithm makes the decision early and change the algorithmic path after decision, and will never reconsider the old decisions. It may not be accurate for some problems.

Optimal Sub structure: A problem exhibits optimal sub-structure, if an optimal solution to the sub-problem contains within its optimal solution to the problem.

3.1.3 Dijkstra's Algorithm description

The algorithm has different states.

In the K state, the shortest routes to the K nearer nodes have been determined and these nodes are within a denominated group M .

In state $K + 1$, a node that is not in M and has the shortest route from the origin node is including in M .

Finally, all the nodes will be in M , and their routes from the origin will have been determined. The algorithm can be described as it follows:

- N = number of nodes in the graph
- s = source node

- M = group of nodes that have been incorporated by the algorithm.
- D_n = cost of the route with the smaller cost, from node s to node n .

The algorithm has three steps; steps 2 and 3 are repeated until $M = N$. Such steps are repeated until the final routes have been assigned to all the nodes of the network:

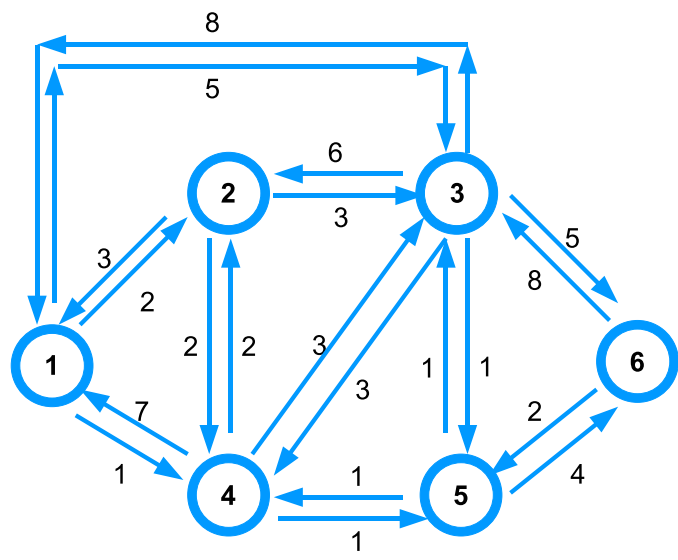
1. To initialize: In the group of nodes M , introduce the node origin. The costs of the initial routes towards the next nodes are simply the costs of the connection.
2. Find the neighbour node that it is not in M and that has the connection with smaller cost from node s . Insert the found node in M .
3. Update the routes with the minimum cost.

Each iteration of steps 2 and 3 get up a new node in M and defines the route with minimum cost from s to that node. That route through only nodes that are in M .

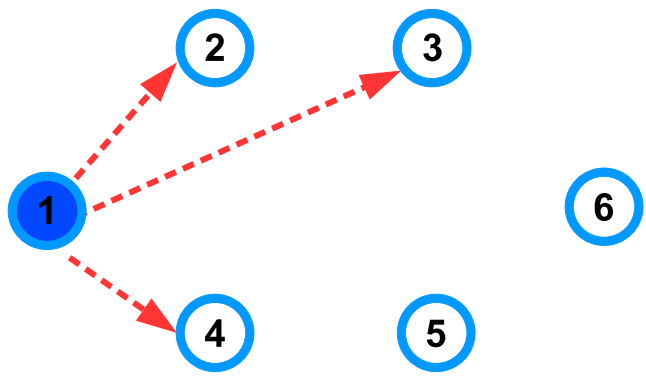
Next, we will analyze two examples to figure out the Dijkstra's algorithm execution.

3.1.4 Dijkstra: Example 1

Graph to analyze:

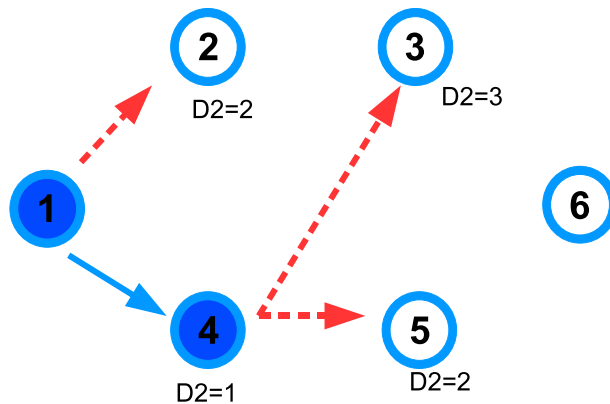


► *Iteration 1*



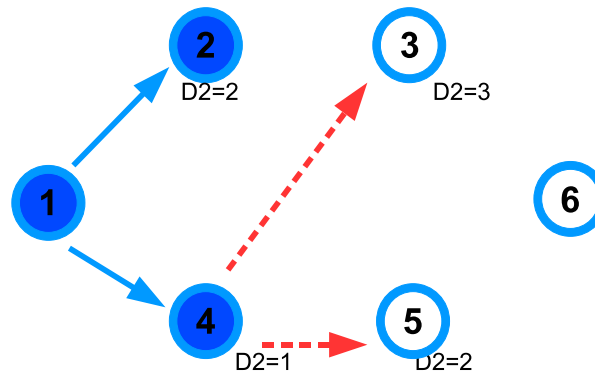
Iteration	M	D2 Route	D3 Route	D4 Route	D5 Route	D6 Route
1	1	2 1,2	5 1,3	1 1,4	- -	- -

Initialization. Insert in M the node origin, and insert the costs of the connections of the node origin.

► *Iteration 2*

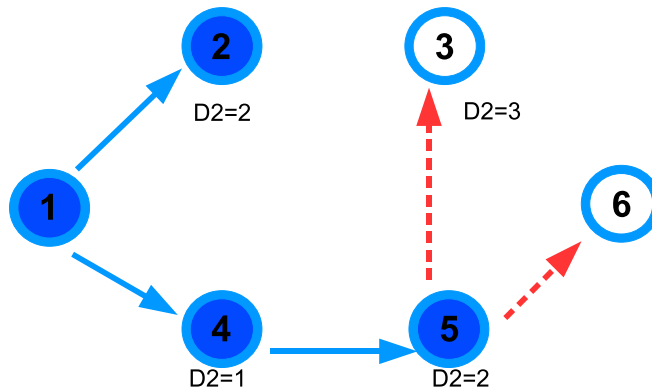
Iteration	M	D2 Route	D3 Route	D4 Route	D5 Route	D6 Route
2	1,4	2 1,2	4 1,4,3	1 1,4	2 1,4,5	- -

Find the neighbour node that it is not in M and that has the connection with smaller cost from node s . In this case the node is 4. Insert the found node in M and update the routes with the minimum cost.

► *Iteration 3*

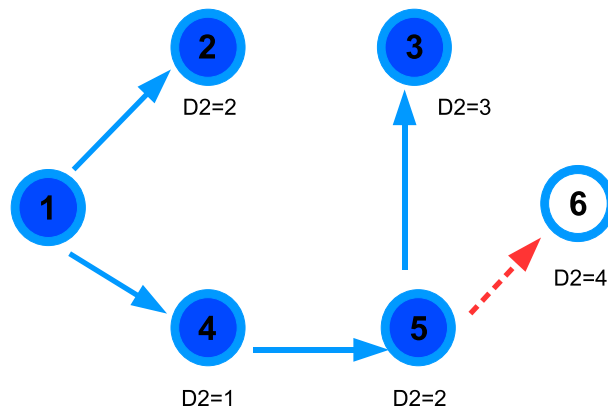
Iteration	M	D2 Route	D3 Route	D4 Route	D5 Route	D6 Route
3	1,2,4	2 1,2	3 1,4,3	1 1,4	2 1,4,5	- -

Find the neighbour node that it is not in M and that has the connection with smaller cost from node s . In this case the node is 2. Insert the found node in M and update the routes with the minimum cost.

► *Iteration 4*

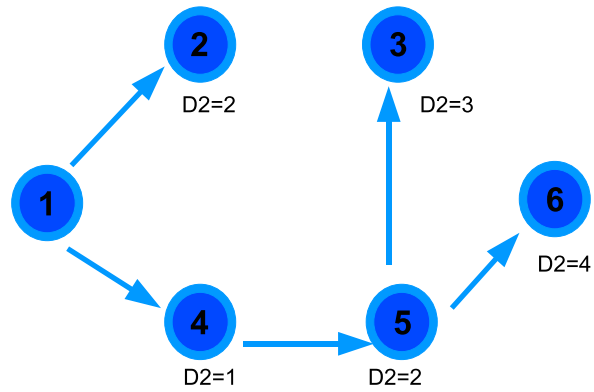
Iteration	M	D2 Route	D3 Route	D4 Route	D5 Route	D6 Route
4	1,2,4,5	2 1,2	3 1,4,5,3	1 1,4	2 1,4,5	4 1,4,5,6

Find the neighbour node that it is not in M and that has the connection with smaller cost from node s . In this case the node is 5. Insert the found node in M and update the routes with the minimum cost.

► *Iteration 5*

Iteration	M	D2 Route	D3 Route	D4 Route	D5 Route	D6 Route
5	1,2,3,4,5	2 1,2	3 1,4,5,3	1 1,4	2 1,4,5	4 1,4,5,6

Find the neighbour node that it is not in M and that has the connection with smaller cost from node s . In this case the node is 3. Insert the found node in M and update the routes with the minimum cost.

► *Iteration 6*

Iteration	M	D2 Route	D3 Route	D4 Route	D5 Route	D6 Route
6	1,2,3,4,5	2 1,2	3 1,4,5,3	1 1,4	2 1,4,5	4 1,4,5,6

Find the neighbour node that it is not in M and that has the connection with smaller cost from node s . In this case the node is 6. Insert the found node in M , and as M is equal than N , it means that we don't have more routes to find and all the nodes are in M so we have finished our search.

3.1.5 Dijkstra: Example 2

We are going to show another trace of Dijkstra's algorithm in this example. This time we are going to follow the pseudo-code step by step¹.

DIJKSTRA'S ALGORITHM - PSEUDO-CODE

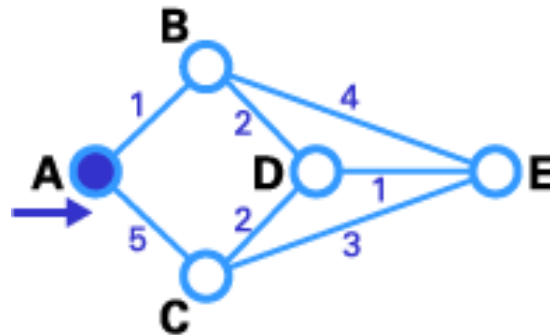
```

1 function Dijkstra(G, w, s)
2   for each vertex v in V[G] // Initializations
3     d[v] := infinity
4   previous[v] := undefined
5   d[s] := 0
6   S := empty set
7   Q := set of all vertices
8   while Q is not an empty set // The algorithm itself
9     u := Extract_Min(Q)
10    S := S union {u}
11    for each edge (u,v) outgoing from u
12      if d[v] > d[u] + w(u,v) // Relax (u,v)
13        d[v] := d[u] + w(u,v)
14        previous[v] := u

```

Initialization

- Set cost of all the vertexes to ∞
- Set cost first node to 0
- ↔ $S = \{ \}$ - Final set
- ↔ $Q = \{ [A, 0], [B, \infty], [C, \infty], [D, \infty], [E, \infty] \}$ - Initial Set
- ↔ $u = CurrentNode$
- ↔ $v = NextNode$
- ↔ $Previous = \{ [A, -], [B, -], [C, -], [D, -], [E, -] \}$ - Info with previous nodes.



¹Accessed 05/03/06: <http://computer.howstuffworks.com/routing-algorithm3.htm>

► *Iteration 1*

Q empty? \rightarrow No

$$u = \{[A, 0]\}$$

$$S = S \cup \{u\} = \{[A, 0]\}$$

For each outgoing edge from u :

Outgoing edge 1:

if $[B, \infty] > [A, 0] + 1$: \rightarrow Yes

$$[B, \infty] = [B, 1]$$

$$\text{Previous}\{B\} = A$$

Outgoing edge 2:

if $[C, \infty] > [A, 0] + 5$: \rightarrow Yes

$$[C, \infty] = [C, 5]$$

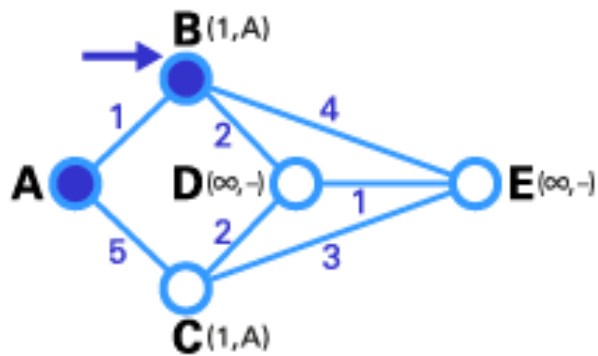
$$\text{Previous}\{C\} = A$$

Current status:

$$\hookrightarrow S = \{[A, 0]\}$$

$$\hookrightarrow Q = \{[B, 1], [C, 5], [D, \infty], [E, \infty]\}$$

$$\hookrightarrow \text{Previous} = \{[A, -], [B, A], [C, A], [D, -], [E, -]\}$$



► *Iteration 2*

Q empty? → No

$$u = \{[B, 1]\}$$

$$S = S \cup \{u\} = \{[A, 0], [B, 1]\}$$

For each outgoing edge from u :

Outgoing edge 1:

if $[E, \infty] > [B, 1] + 4$: → Yes

$$[E, \infty] = [E, 5]$$

$$\text{Previous}\{E\} = B$$

Outgoing edge 2:

if $[D, \infty] > [B, 1] + 2$: → Yes

$$[D, \infty] = [D, 3]$$

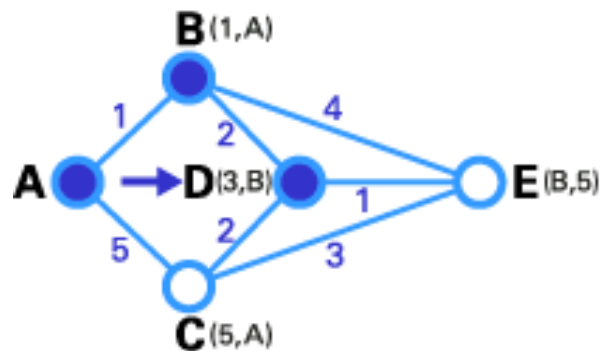
$$\text{Previous}\{D\} = B$$

Current status:

$$\hookrightarrow S = \{[A, 0], [B, 1]\}$$

$$\hookrightarrow Q = \{[C, 5], [D, 3], [E, 5]\}$$

$$\hookrightarrow \text{Previous} = \{[A, -], [B, A], [C, A], [D, B], [E, B]\}$$



► *Iteration 3*

Q empty? → No

$$u = \{[D, 3]\}$$

$$S = S \cup \{u\} = \{[A, 0], [B, 1], [D, 3]\}$$

For each outgoing edge from u :

Outgoing edge 1:

if $[E, 5] > [D, 3] + 1$: → Yes

$$[E, 5] = [E, 4]$$

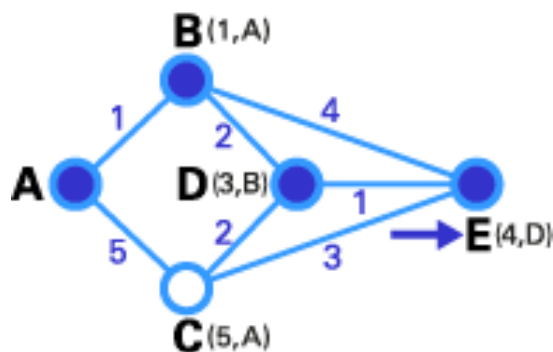
$$Previous\{E\} = D$$

Current status:

$$\hookrightarrow S = \{[A, 0], [B, 1], [D, 3]\}$$

$$\hookrightarrow Q = \{[C, 5], [E, 4]\}$$

$$\hookrightarrow Previous = \{[A, -], [B, A], [C, A], [D, B], [E, D]\}$$



► *Iteration 4*

Q empty? → No

$$u = \{[E, 4]\}$$

$$S = S \cup \{u\} = \{[A, 0], [B, 1], [D, 3], [E, 4]\}$$

For each outgoing edge from u : → None

Current status:

$$\hookrightarrow S = \{[A, 0], [B, 1], [D, 3], [E, 4]\}$$

$$\hookrightarrow Q = \{[C, 5]\}$$

$$\hookrightarrow Previous = \{[A, -], [B, A], [C, A], [D, B], [E, D]\}$$

► *Iteration 5*

Q empty? → No

$$u = \{[C, 5]\}$$

$$S = S \cup \{u\} = \{[A, 0], [B, 1], [D, 3], [E, 4], [C, 5]\}$$

For each outgoing edge from u :

Outgoing edge 1:

$$\text{if } [D, 3] > [C, 5] + 2: \rightarrow \text{No}$$

Outgoing edge 2:

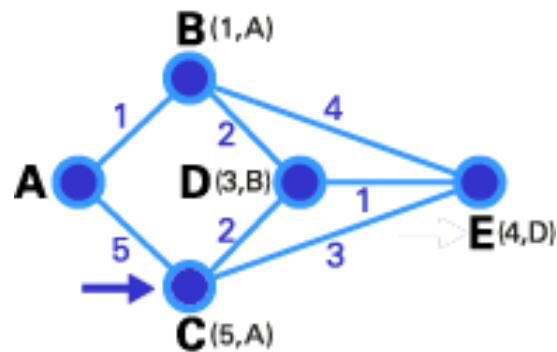
$$\text{if } [E, 4] > [C, 5] + 3: \rightarrow \text{No}$$

Current status:

$$\hookrightarrow S = \{[A, 0], [B, 1], [D, 3], [E, 4], [C, 5]\}$$

$$\hookrightarrow Q = \{\}$$

$$\hookrightarrow \textit{Previous} = \{[A, -], [B, A], [C, A], [D, B], [E, D]\}$$



► *Iteration 6*

Q empty? → Yes

End Dijkstra

3.1.6 Related problems, applications and algorithms

The functionality of Dijkstra's original algorithm can be extended with a variety of modifications. For example, sometimes it is desirable to present solutions which are less than mathematically optimal. To obtain a ranked list of less-than-optimal solutions, the optimal solution is first calculated. A single edge appearing in the optimal solution is removed from the graph, and the optimum solution to this new graph is calculated. Each edge of the original solution is suppressed in turn and a new shortest-path calculated. The secondary solutions are then ranked and presented after the first optimal solution.

OSPF (open shortest path first) is a well known real-world implementation of Dijkstra's algorithm used in internet routing.

Unlike Dijkstra's algorithm, the **Bellman-Ford** algorithm can be used on graphs with negative edge weights, as long as the graph contains no negative cycle reachable from the source vertex s . (The presence of such cycles means there is no shortest path, since the total weight becomes lower each time the cycle is traversed.)

A related problem is the traveling salesman problem, which is the problem of finding the shortest path that goes through every vertex exactly once, and returns to the start. This problem is NP-hard; in other words, unlike the shortest path problem, it is unlikely to be solved by a polynomial-time algorithm.

If additional information is available that estimates the "distance" to the target, the **A*** algorithm can be used instead to cut down on the size of the subset of the graph which is explored.

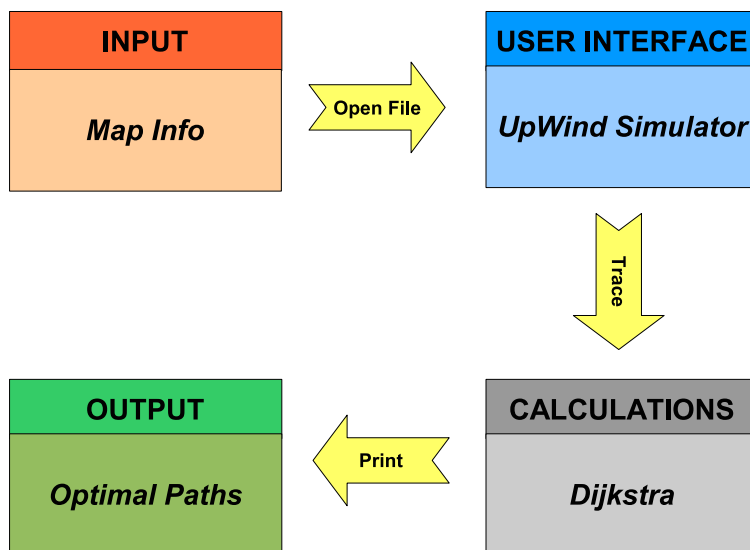
UpWind Simulator

Contents

4.1	Input: Map file format	29
4.2	Upwind Simulator: User Interface	30
4.2.1	Main form	30
4.2.2	File Menu	31
4.2.3	System Menu	32
4.3	Processing info & algorithm implementation	33
4.3.1	Objects definition	33
4.3.2	Whole process overview	34
4.3.3	Data parsing	35
4.3.4	Drawing map info	37
4.3.5	Drawing and storing control points	38
4.3.6	Handling information and performing calculations	38
4.3.7	Result: Optimal path	55

We thought that was absolutely necessary to test all the algorithms that we had planned to implement, so we would develop a simulator to analyze whether everything worked in the way that we had planned.

The following chart represents the simulator data flow:



The main information is shown in the chart. As we can appreciate in it, we can divide the data flow in four different parts: input, user interface, calculation and final result.

Explaining the chart parts with another words, the user interface had to be able of open a map file with information about islands, and display the information on the screen. Once we had everything drawn on the screen the user had to interact with the application and select the origin and destination points, press trace button and it would show the final result: optimal paths.

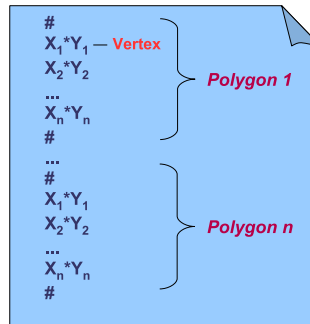
4.1 Input: Map file format

To test our algorithm and application we thought about the kind of input that we had to provide to the application.

The most suitable information would be ESRI files, but to open ESRI files was not the goal of this project, somehow we had to manage to calculate the shortest path between at least two points using Dijkstra's algorithm using real information provided by the maps.

The Dijkstra's algorithm would need polygonal information about the islands so we thought about defining our own file format to handle island information easily.

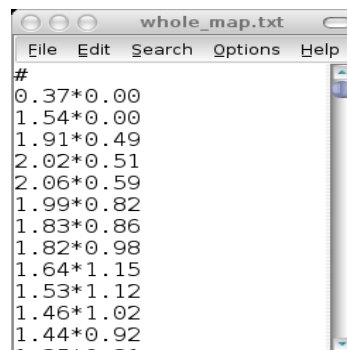
The map file format that we defined is as follows:



As its shown in the image, the file format is very simple. We defined polygons using a set of vertexes so each polygon has "n" vertexes. To separate each polygon we used # symbol. To avoid errors this is the formal regular expression used to define the map file format:

$$(\#\backslash n(\textit{number}\backslash * \textit{number}\backslash n)^2(\textit{number}\backslash * \textit{number}\backslash n)^+ \#\backslash n)^+$$

This is an example of a real map file prototype:



4.2 Upwind Simulator: User Interface

Once we defined our map file format we created the application which had to be able to handle the map data and display it.

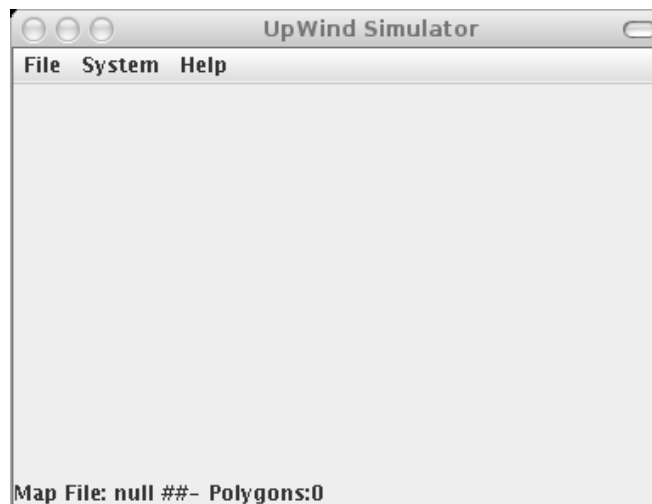
To implement the simulator we used Java programming language because its object oriented structure would make easier our task. The following open source tools were used too:

- Eclipse 3.1
- Eclipse Plug-in: Visual Editor

We didn't want to implement a difficult application. It would be just a simulator so we implemented just the features we thought that were useful.

4.2.1 Main form

Next is shown a screenshot of the main application form:



The application is composed by the main menu bar and the information bar placed on the top and bottom.

The menu bar is subdivided in 3 submenus: File, System and Help. The user will use them to interact with the application.

In the information bar will be shown the errors, number of polygons loaded in the screen and optimal paths found.

4.2.2 File Menu

At the same time, the "File Menu" is formed by 3 items: Open Map File, Reset Map Info and Exit.

Open Map File: Using this button, the user will be able to open map files. The user will have to chose the right file and it will be shown automatically on the screen.



Reset Map Info: Pressing this button: control points, visibility graph and paths will be deleted automatically from the map. It lets us work with the same map more times without reopen it.

Exit: By using this button, the user will be able to close the simulator.

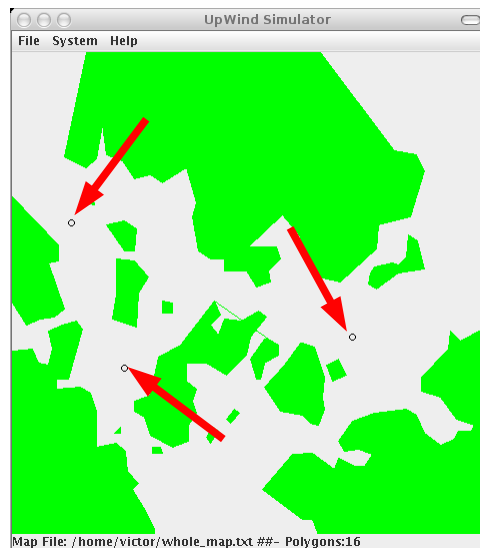
4.2.3 System Menu

The "System Menu" is formed by 5 items: trace routes, build visibility graph, add control point, enable zoom and enable panning.

Trace Routes: Using this button, the user will be able to calculate visibility graph and Dijkstra's algorithm getting the optimal paths as a result the both executions. Only the optimal paths will be shown on the simulator screen.

Build visibility graph: Using this button, the user will build only the visibility graph and after the calculations it will be shown on the screen.

Add control point: The user will add control points into the map pressing this button. Before calculating the visibility graph is absolutely necessary to add at least 2 control points. The user will use left mouse button to add the points.



Enable zoom: When this option is enabled, is possible to zoom in and zoom out the map by using the mouse scroll wheel.

Enable panning The user will be able to move the whole map using drag and drop using the left mouse button.

4.3 Processing info & algorithm implementation

In this part we are going to explain everything related with the "UpWind Simulator" implementation. We will start explaining the main objects definition and afterwards we will follow the application execution and we will comment all the essential information.

4.3.1 Objects definition

We already told that we would have to provide geometrical information to Dijkstra's algorithm. The main geometrical object we had was the "Polygon". It was clear that the object Polygon would be one of the most important in our implementation. So we defined the object "CPolygon" who was formed by edges and vertexes whose we created "CEdge" and "CVertex" class too.

The next chart represents the class diagram, in this chart we'll show only the basic information, it is not the final class diagram:

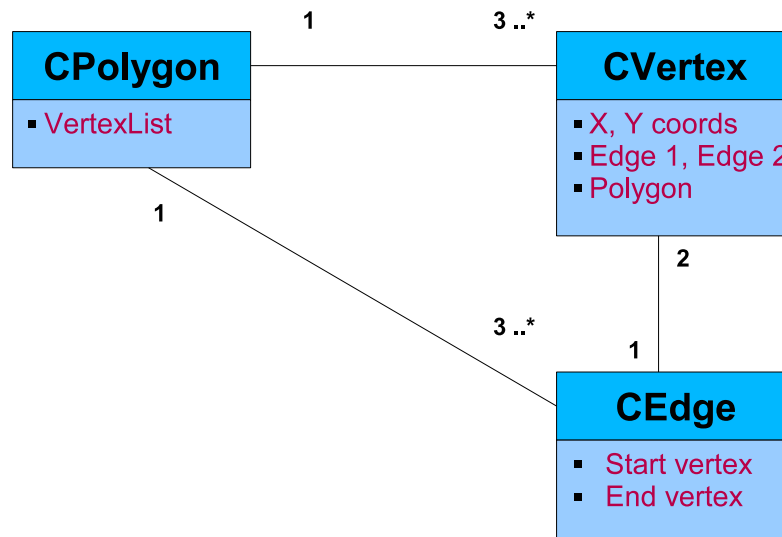
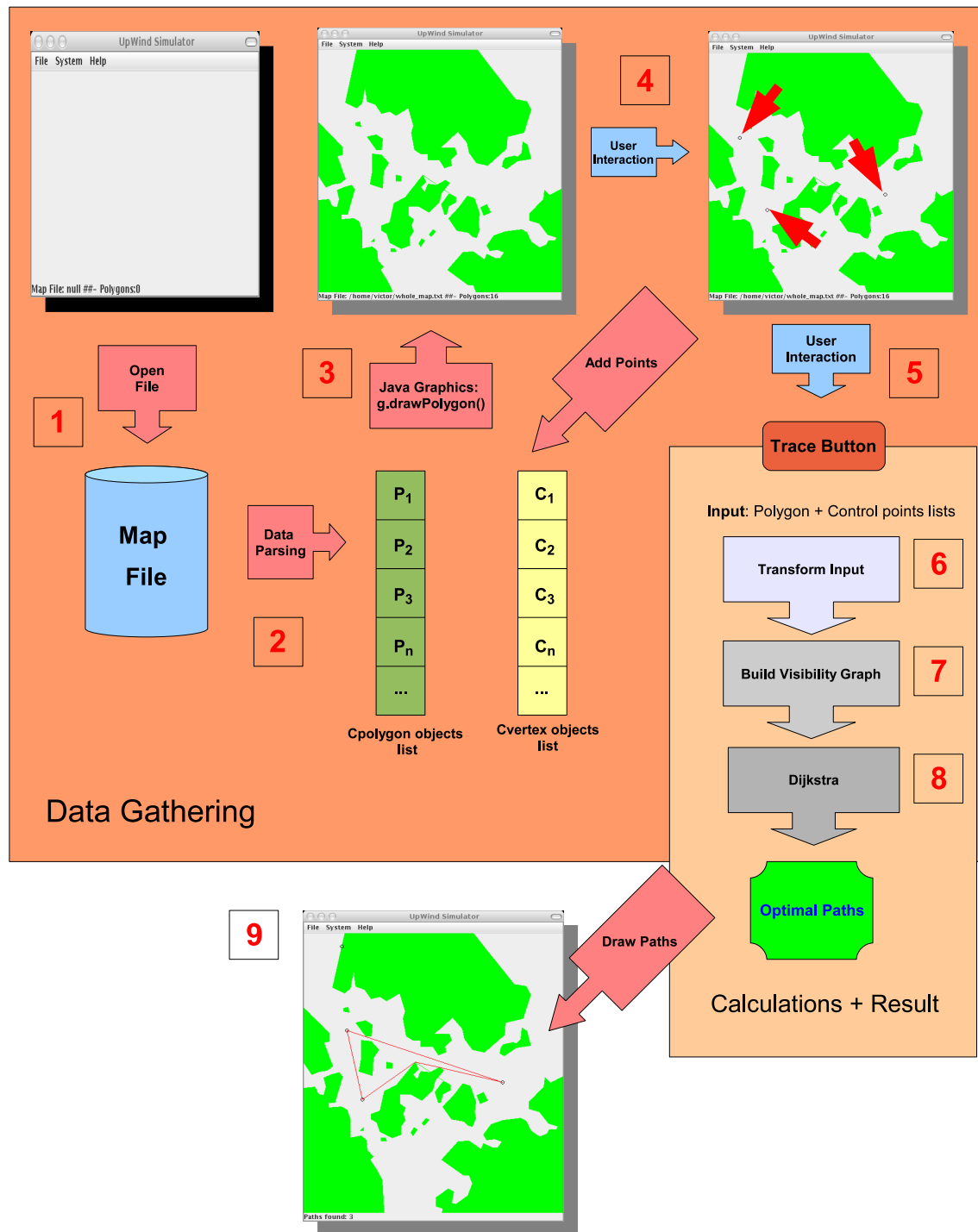


Chart reading: One polygon is formed for at least three vertex or at least three edges. One vertex has coordinates x and y, belongs to a polygon and belongs to two different edges. One edge is composed by two vertexes.

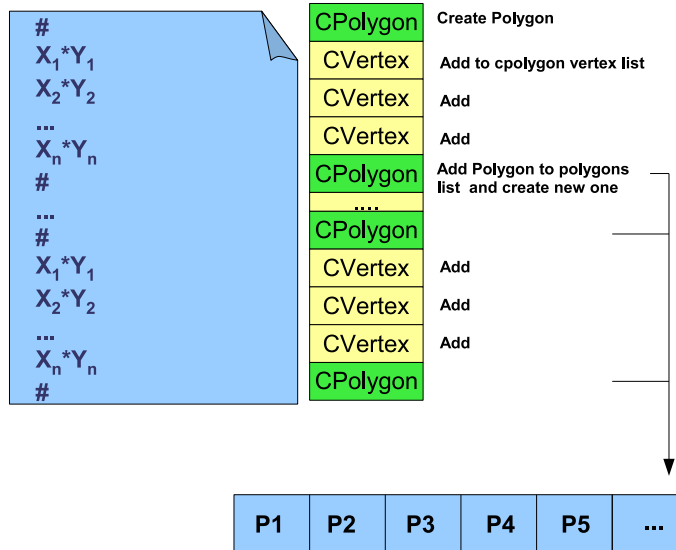
4.3.2 Whole process overview

The explanation of the whole process is going to be hard and difficult. It will be really easy to lose the track about where we are and where do we want to go. Therefore we have thought to draw a chart giving you a global view of the whole process.



4.3.3 Data parsing

The first user interaction consists in opening one map file. The user will select the desired file and internally, the application will parse the file map. As we said before, everything between two '#' symbols will be one CPolygon object which will be composed by 'n' CVertex objects. Each time we find a CPolygon object, we will store it in a CPolygon objects list.



Below is shown the the parsing method source code:

```

MAP DATA PARSING - LOADMAP [MAP.JAVA]
1 private void LoadMap() {
2
3     CPolygon p = null;
4     double x,y;
5     CVertex v = null;
6     x=y=0.0;
7     int npols = 0;
8     int npoints = 0;
9     try {
10        BufferedReader mapfile = new BufferedReader(
11            new FileReader(MapName));
12        String line;
13        while ((line = mapfile.readLine()) != null) {
14            if (line.equals("#") && npols == 0) { //First line
15                p = new CPolygon();
16                npols++;
17            } else if (line.equals("#") && npols > 0){
18                //End polygon and beginning new one
19                vPolygons.addElement(p);
20                p = null;
21                p = new CPolygon();

```

```
22         npols++;
23     } else if (!line.equals("#")) { //vertex detected
24         String[] parsed = line.split("\\*");
25         if (parsed.length == 2) {
26             x = Double.parseDouble(parsed[0]);
27             y = Double.parseDouble(parsed[1]);
28             v = new CVertex(x,y);
29             p.AddVertex(v);
30             npoints++;
31         }
32     } else
33         System.out.println("Error, _reading_sth_unknown");
34     }
35     mapfile.close();
36 } catch (IOException e) {}
37 }
```

4.3.4 Drawing map info

Once we have all the polygonal info stored in the polygon list, we have to draw it in the main application form. To perform the drawing action we use the java graphical library. This library is able to draw polygons on the screen using java polygons objects and the method:

- *g.drawPolygon(Polygon p);*

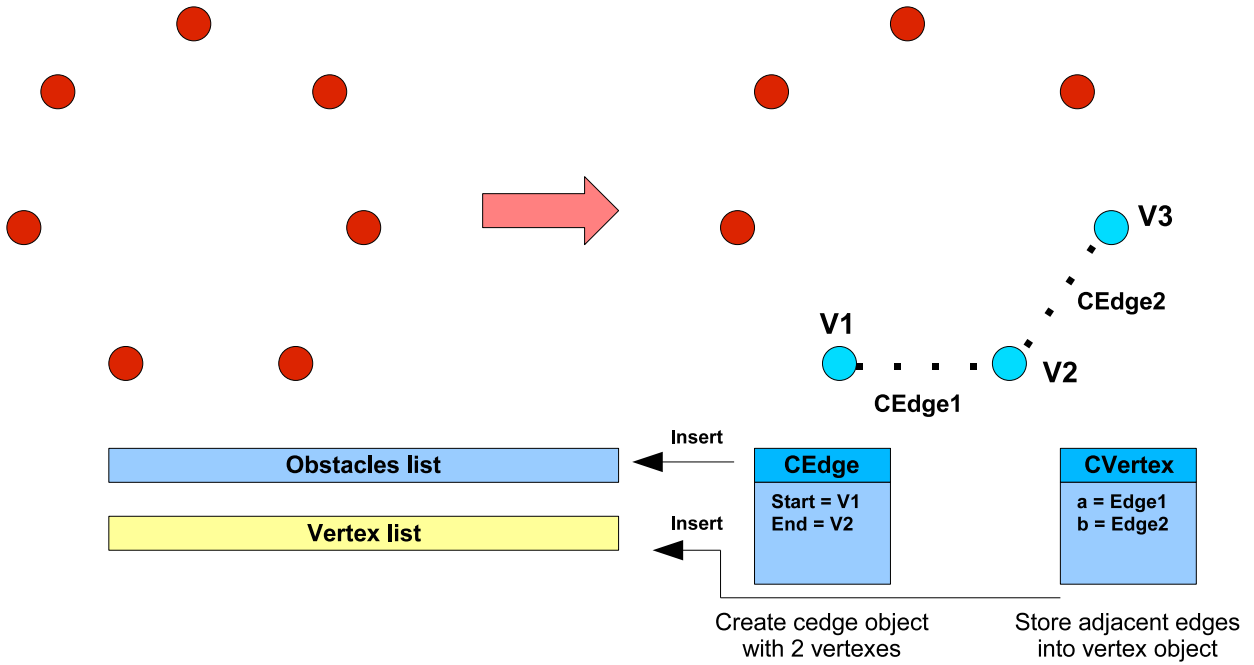
The following step will be transform our CPolygon objects into Java Polygon objects once we have the transformation done, we will execute the above function to draw in the screen.

```

DRAWING MAP - DRAWMAP [MAP.JAVA]
1 public void DrawMap(Graphics g) {
2
3     for (int i = 0 ; i < vPolygons.size() ; i++) {
4         CPolygon cp = (CPolygon) vPolygons.elementAt(i); //CPolygon
           object
5         Polygon p = new Polygon(); // Java Polygon object
6         int a,b;
7         int zoom = 200;
8         int translation = 0;
9         for(int j = 0 ; j < cp.vertexList.size() ; j++) {
10            CVertex cv = (CVertex) cp.vertexList.elementAt(j);
11            a = (int) (cv.x * zoom) + translation;
12            b = (int) (cv.y * zoom) + translation;
13            p.addPoint(a,b); //Add vertex to Polygon
14            g.setColor(Color.RED);
15            g.setColor(Color.GREEN);
16            g.fillPolygon(p);
17        }
18        g.drawPolygon(p); //Draw Java Polygon
19    }
20 }

```

The transformation is very simple: We scan the polygon list and we stop in every CPolygon object. Inside each CPolygon object there is a vertex list, we access to each vertex inside this list and we add the vertex coordinates to the new java Polygon object. When we have scanned all the vertexes we fill and draw the java polygon object on the screen.



We will scan all the polygons of the polygon list. We will stop in each one and we will scan its vertex list. We will use 3 vertexes to get 2 edges: $i, i + 1$ and $i - 1$

Iterative process for all the vertexes:

1. Cedge1 = i and $i - 1$ vertexes.
2. Cedge2 = i and $i + 1$ vertexes.
3. Store cedges info in vertex i .
4. Add vertex i in vertex list.
5. Add Cedge1 in obstacle list.

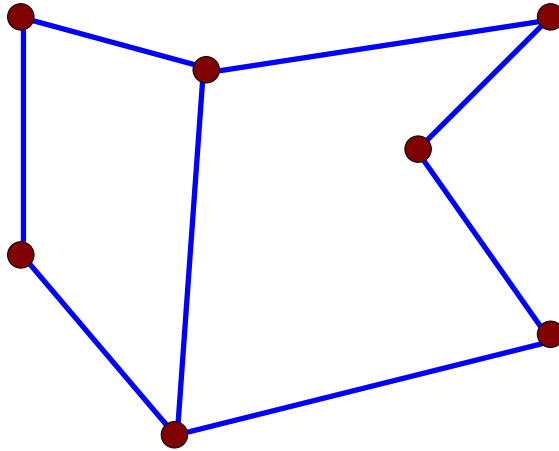
We will put out of this iteration special cases like first and last vertexes. ¹

¹See source code: Build method [CVisibilityGraph.java]

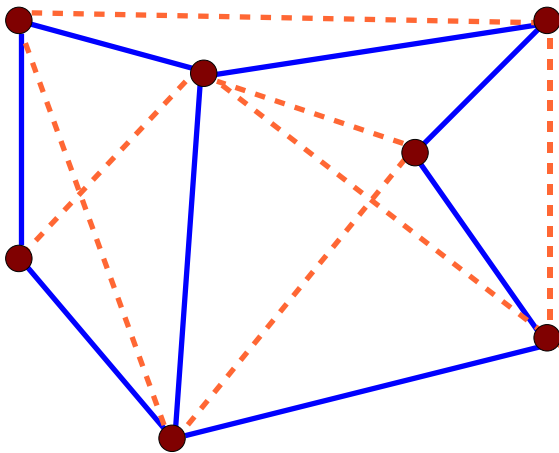
Building Visibility Graph

Firstly, we have to know what is a visibility graph and how to build it.

A *graph* is a set of objects called vertices (or nodes) connected by links called edges (or arcs), which can also have associated directions. Typically, a graph is depicted as a set of dots (i.e., vertices) connected by lines (i.e., edges), with an arrowhead on a line representing a directed arc.

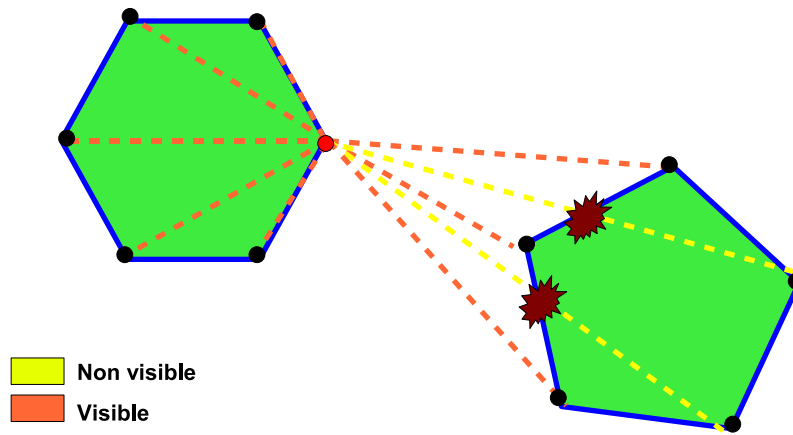


A *visibility graph* is a graph of intervisible locations. Each node or vertex in the graph represents a point location, and each edge represents a visible connection between them (that is, if two locations can see each other, an edge is drawn between them).

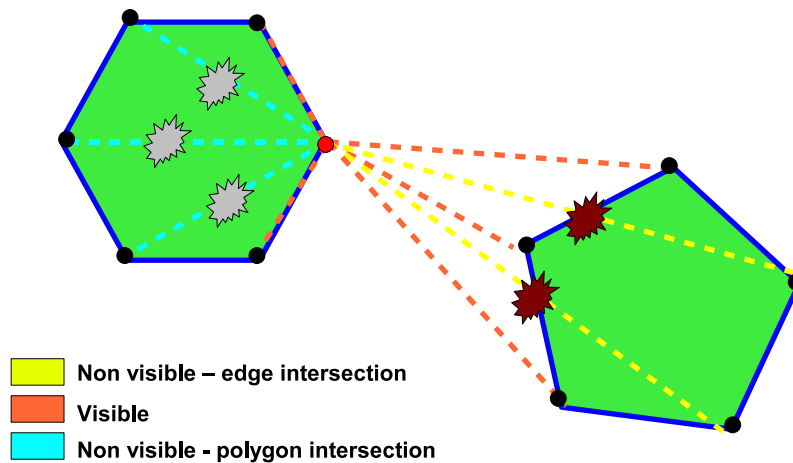


Once the visibility graph concept is clear, we have to place the visibility graph into our problem. In our problem definition we have isolated polygons (each one represented one island), and we have to build a visibility graph joining all the visible vertexes.

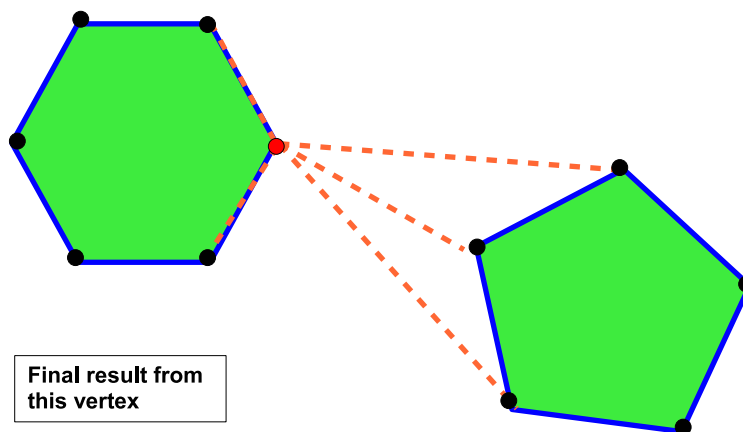
We're going to analyze the drawing below to understand better our current problem. We'll analyze the situation for one single vertex:



This is the result of launching rays between every single vertex in both polygons. As you can see there are two rays intersecting with the edges of the polygon, so we will delete them from the graph.



We don't have to forget that our polygons are islands and we don't want to go through them. On the left polygon we can see that there are some visible vertices inside it, we have to remove them from the polygon. It's absolutely necessary don't add the inner edges into the visibility graph.



Finally, these are the visible vertexes from the analyzed vertex. Understand the concept is quite easy, the difficult work is to apply the right methods to do it possible. Next we will explain all the process to built the visibility graph.

The process of building the visibility graph is hard and tedious, we have had serious problems to build the right graph. We will comment all the main process and afterwards we will comment the problems we had and how we solved them.

The main idea is shown in the drawing above. We have to go through every single vertex in the graph and create rays between all the vertexes. We will check whether there is something intersecting to this ray or virtual edge. Whether there is nothing intersecting this ray we will add it to the final obstacle list otherwise, we will skip it.

We will follow the next steps:

- Select virtual edge.
- Check whether it has intersections with other edges.
- if there is intersection:
 - Skip this edge
- else: (there is not intersection)
 - if it is not a inner diagonal:
 - * Add to final edgelist = Visibility Graph
 - else: (it's inner diagonal)
 - * Skip this edge

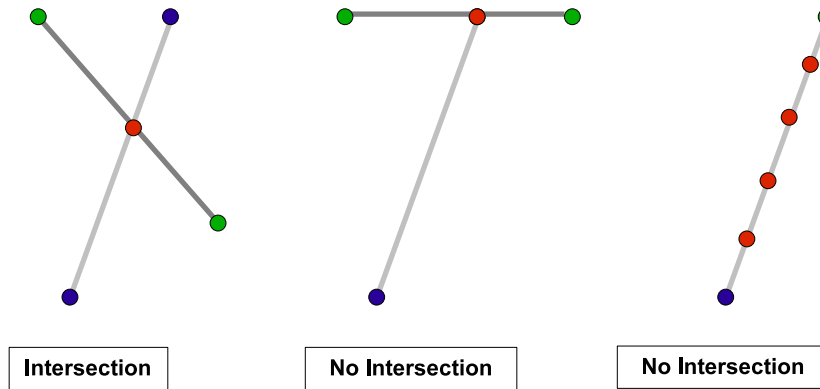
Intersection tests

In our last step we filled two lists, vertex and obstacle list. Now, we are going to scan all the elements in the vertex list and we will create virtual edges. Each edge has to be subjected to some intersection tests. To do this we will check if the edge intersects with any edge of each polygon of our graph.

The intersection test is a mathematical operation. Mathematically is possible to know if two rules or segments are intersected so we found one algorithm [1] to do this checking.²

The algorithm checks if there is a point in common between the two segments. This algorithm will return INTERSECTION while working with crossed lines. If both lines are only intersected for one side vertex or they are the same segment, one over the other one, it will return "NO INTERSECTION". The polygon edges will take part in our visibility graph, therefore the intersection test will return false when checking two overlapped segments.

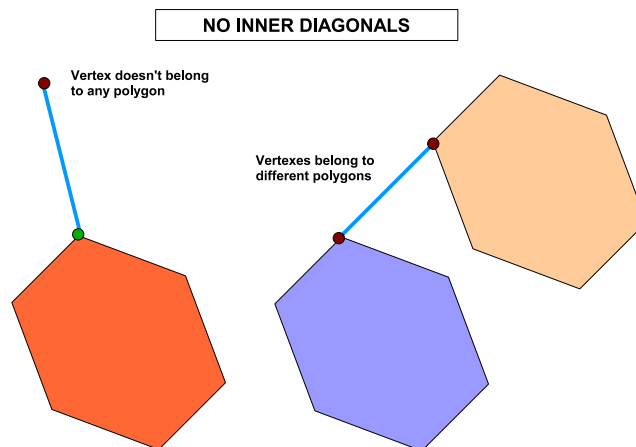
²Accessed 13/02/06: http://softsurfer.com/Archive/algorithm_0108/algorithm_0108.htm

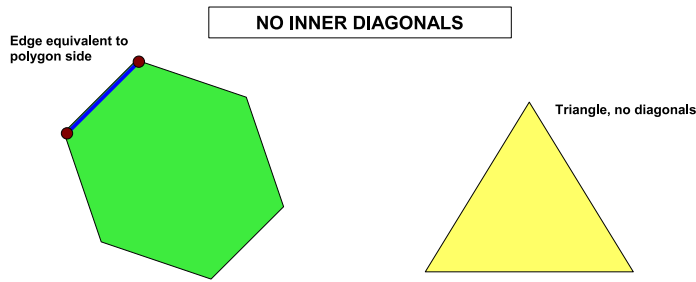


Inner diagonal tests

When one edge succeeds the intersection test other test is waiting for it. We will add it to the edgesList(Visibility Graph) only if it is not an inner diagonal of one polygon. We can skip a lot of edges and calculation time with two simple checking before subjecting it to the hard and tedious test of the inner diagonal.

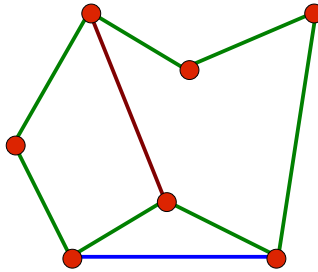
- If one of the both vertexes of the edge doesn't belong to any polygon. It isn't an inner diagonal.
 - Add to edge list.
- If both vertexes belong to different polygons. It isn't an inner diagonal.
 - Add to edge list.
- If the polygon has 3 vertexes (triangle), it hasn't got diagonals.
 - Add to edge list.
- If the edge is equivalent to one polygon edge. It isn't a diagonal.
 - Add to edge list.
- Otherwise: Check if it is inner diagonal.





Inner diagonal:

We have ruled out a lot of possibilities but we still have to do the last test because we can find edges placed like the following one and we have to check if it is outside or inside the polygon:



As you can see in the drawing above, there are two lines, red and blue. We have to be able of detect that the red one is inside the polygon and the blue one is not inside.

We will apply the cross-line method, this method is one of the most commons in this kind of problems. We will explain method step by step.

We have one edge and we would like to know whether it is inside the polygon or not. The cross-line [1] method will determine whether a point is inside a complex polygon. It will work with points so we have to extract a point from the edge. The middle point. Once we have the point we want to work with we have to give it to the cross-line algorithm. It will work in that way.³

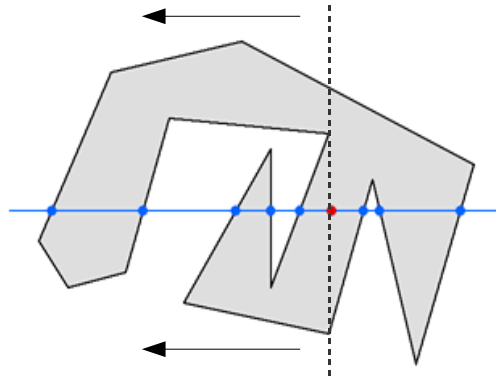
The main idea is to launch a ray breaking all the polygon. We have to scan all the edges of the polygon and determine which one of the vertical edges are crossing the ray. Each time we detect one vertical edge crosses the ray and the x coordinate of the edge is less than x coordinate of the middle point, we will increment one counter variable. Finally, after scanning all the edges in the polygon we will check the crossing counter. if its is odd the point will be inside otherwise it will be outside.

Let's make a visual test with these drawings:

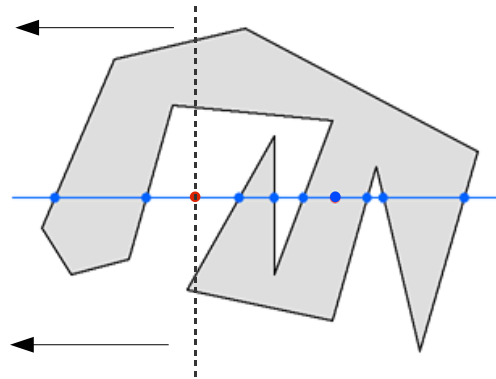
³Accessed 23/04/06:

<http://www.aliendyflex.com/polygon/>

http://softsurfer.com/Archive/algorithm_0103/algorithm_0103.htm



As we can see in the above figure, if we count the crossing lines on the left of the middle point we will check that the number is odd. Number of crossings: 5, point inside.



Another example shows. Number of crossings: 2, even, point outside.

The source code we used is the next one:

```

POINT INSIDE POLYGON - 1ST VERSION [CPOLYGON.JAVA]
1  boolean Inside(CVertex cvertex) //Input: middle point
2  {
3      double d = cvertex.x;
4      double d1 = cvertex.y;
5      int i = vertexList.size();
6      int j = 0;
7      //Scan all the vertexes
8      for(int k = 0; k < i; k++)
9      {
10         CVertex cvertex1=(CVertex) vertexList.elementAt(k);
11         CVertex cvertex2=(CVertex) vertexList.elementAt(((k+i)-1)%i);
12         CVertex cvertex3=(CVertex) vertexList.elementAt((k+1)%i);
13         CEdge cedge = new CEdge(cvertex1, cvertex3);
14
15         //Check whether the cedge is vertical and crossing the ray
16         if( (cedge.start.y > d1 && cedge.end.y < d1) ||
17             (cedge.start.y < d1 && cedge.end.y > d1) )
18         {
19             //Get the intersection xcoordinate between cedge and
20             ray
             double d2 = cedge.GetX(d1);

```

```

21             //Check if the cedge is on the left of the middle
                point
22             if(d2 < d)
23                 j++;
24         }
25     }
26
27     //return:     true , odd
28                 false , even
29     return j % 2 == 1;
30 }

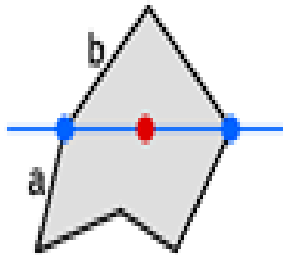
```

PROBLEM DETECTED!

After testing the algorithm, we observed that it didn't work as we expected. The result was really strange and we had to study a lot of examples to try to isolate the error and guess what was going on and which were the reasons. We will describe next the error.

We will use these figures to show in which simple cases it worked in the wrong way.

Example 1



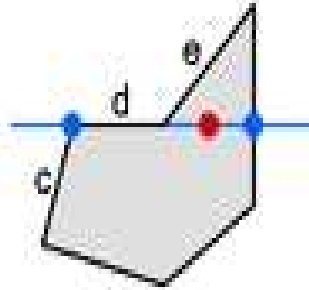
Following the algorithm execution, the conditional we should evaluate to know whether the point is inside or not is the following one:

```

...
ymp = d1; // y coordinate middle point
if( (cedge.start.y > ymp && cedge.end.y < ymp) ||
    (cedge.start.y < ymp && cedge.end.y > ymp) )
...

```

As is shown in the above figure, the start vertex from the 'a' edge as well as the end vertex from the 'b' edge are the same vertex and it is collinear to the middle point. They are vertical edges but referring to the above conditional test, both of them will skip the conditional without incrementing the crossing counter. It means that after scanning all the edges the counter will show '0'. 0 is consider like even number so the result will be OUTSIDE! and the point is INSIDE!.

Example 2

This is another example which breaks the algorithm. In this example is easy to see that the problem is related with the collinearity too. The whole 'd' edge is collinear to the middle point.

The 'c' edge is not consider crossing the y threshold so it won't increase the counter. The 'd' edge is horizontal and the 'e' edge is not consider crossing the y threshold neither so the result will be '0' again. Point OUTSIDE the polygon!.

The next one is the solution we implemented to solve this problem. We treated vertexes within the y threshold as special cases. This is the source code we add to our algorithm.

```

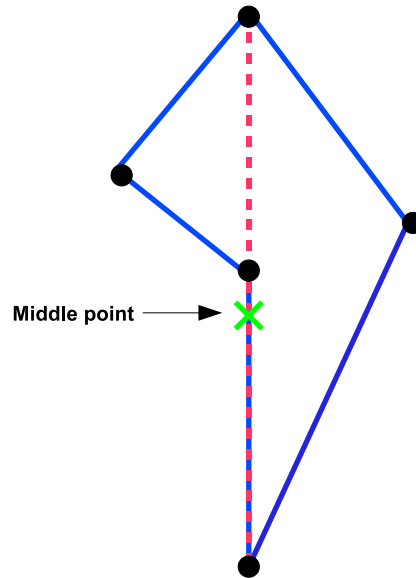
POINT INSIDE POLYGON - IMPROVED VERSION[CPOLYGON.JAVA]
1  boolean Inside(CVertex cvertex) //Input: middle point
2  {
3      ...
4      for(int k = 0; k < i; k++)
5      {
6          ...
7          ...
8          //Special case: Middle point same height than vertex
9          if(d1 == cvertex1.y)
10         {
11             if(cvertex1.x < d){ // Test only left side of the center point
12                 if( (cvertex2.y >= d1 && cvertex3.y < d1) ||
13                     (cvertex2.y < d1 && cvertex3.y > d1) ){
14                     j++;
15                 }
16             }
17         }
18         else if ...
19         {
20             ...
21         }
22     }
23     ...

```

PROBLEM DETECTED!

After solving the last problem, we go on doing more and more test to our building process; the visibility graph have to be perfectly built.

Testing the algorithm we realized it didn't work fine in all the cases. Sometimes, the algorithm did not skip some rules, they were still in the final graph and they should not be. Analyzing a lot of different cases and trying to reduce the error definition we concluded that the problem was linked with collinear points. Let's analyze some examples in the error context.



The red dashed line should be clearly skipped from the visibility graph because it's going through the polygon. Let's trace the situation through the algorithm execution to figure out what is going on in this error example.

The first test that this rule should pass is the "intersection test". As we can see in the above figure, the rule is clearly not crossing to any other edge of the polygon so the intersection test will return, "no intersection".

- Does the rule have intersections with polygon edges? → NO

The rule won't be still skipped from the visibility graph. The next step is checking if the rule is an inner diagonal of the polygon but before as we described previously, it should pass another alternative tests:

- Does the rule have one vertex outside the polygon? → NO
- Does the rule have both vertexes belonging to different polygons? → NO
- Is the rule an polygon edge? → NO


```

37         }
38     }
39     return false;
40 }

```

The code above checks if one vertex is collinear to the polygon. Examining the code, we can see that the rule equation has been used to do the test.

Being:

x, y coordinates of the tested vertex

$x_1, y_1 \rightarrow$ start edge vertex

$x_2, y_2 \rightarrow$ end edge vertex

$$((x_2 - x_1) * y - (y_2 - y_1) * x) == ((x_2 - x_1) * y_1 - (y_2 - y_1) * x_1)$$

If the vertex is collinear both side of the equality will be the same and then the code will proceed to check whether the point is inside the edge extremes.

Once the concept of collinearity is clear, we will comment in which point of the code we will insert it.

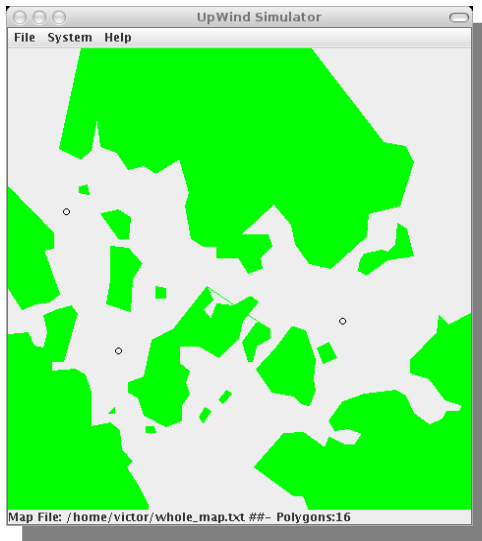
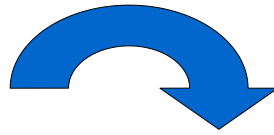
We will do this last test just after checking whether there is intersection or not. If there are not intersections we will store in one edge attribute the result of the collinearity function. Then, after doing all the checking we have been talking above we will check if this attribute is true or false, whether it is false we will add the edge to the final visibility graph list, if it is true, we will skip it.

Coming back to the last example, we will realize that if we subject the edge to the collinearity test, it will find one vertex inside the edge so the method will return true and will skip the edge.

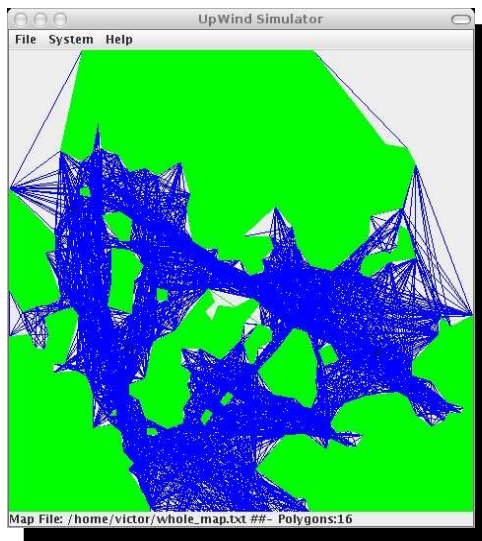
- Does it have any collinear vertex? \rightarrow YES \rightarrow ♣ Skip it! ♣

Finally, we have the visibility graph perfectly built. We will sum up the entire process.

1. We had an input composed by two lists, polygon and control points.
2. We handled these two list in order to build the visibility graph.
3. We did test to assure the right graph building.
4. Visibility Graph already built.



INPUT



VISIBILITY GRAPH

Dijkstra's Algorithm

Everything is ready for the final step. Calculate the optimal paths using Dijkstra's Algorithm. Once the visibility graph is perfectly built, we will implement Dijkstra's algorithm. These will be the steps to follow:

1. Initializations
2. Calculations
3. Get optimal path

Initializations

After getting the visibility graph in the last step, there are still same tasks to do with it. It isn't still ready. As you know the input of the algorithm consists on a *weighted directed graph* and our graph is not weighted yet, so we still have to weigh all the edges of the graph.

Our weighted method is related to the length of each edge.

Coming back to the goal of this project, we should say that the final goal is not just to avoid islands, there are other factors to take into account: speed, wind, another boats, etc... We don't forget about these final goals furthermore we want to leave the project open to implement more and more features to get the final application. At this point, we are calculating the weights of the edges related by the length but whether we try to do more and more complex the calculation cost method, we will get the optimal path without re-programming the application. Everything has been thought to be flexible, so this point will be really important for taking more new factors into account in the future.

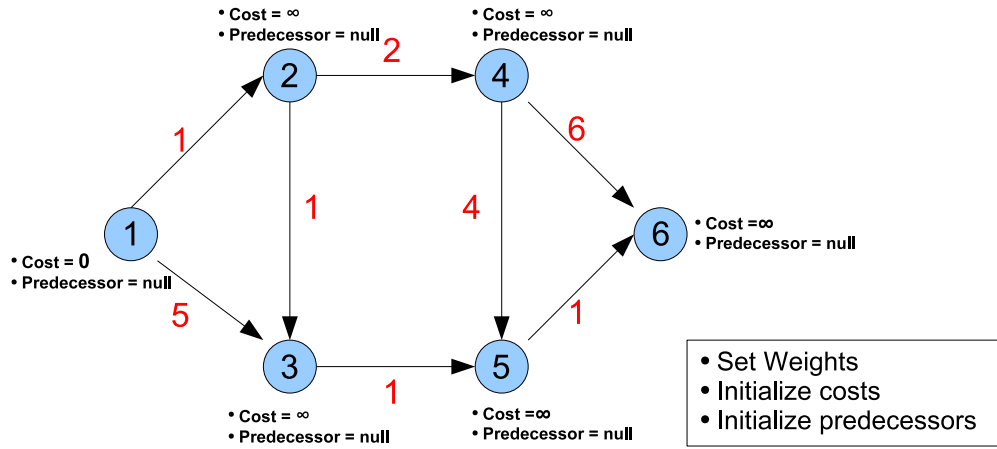
Now, we have to scan every edge of the graph and calculate the weigh using the Euclidean distance formula:

The Euclidean distance between two points $P = (p_1, p_2, \dots, p_n)$ and $Q = (q_1, q_2, \dots, q_n)$ in Euclidean n -space, is defined as:

$$\sqrt{(p_1 - q_1)^2 + (p_2 - q_2)^2 + \dots + (p_n - q_n)^2} = \sqrt{\sum_{i=1}^n (p_i - q_i)^2} \quad (4.1)$$

We have defined Dijkstra to calculate the optimal path between 'n' control points. Now, before calculating Dijkstra we should finish its initialization.

- Create reminder list.
- Set cost of all the nodes to infinity.
- Set cost of the first node to '0'.
- Set predecessor of all the nodes to "null".



Calculations

Everything is ready now for computing the paths, we will follow Dijkstra's algorithm, this is the pseudo-code:

DIJKSTRA'S ALGORITHM [DIJKSTRA.JAVA]

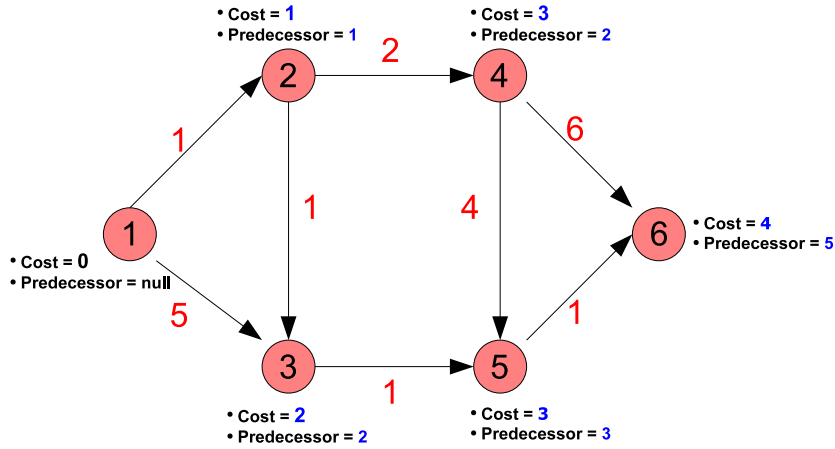
```

1 function Dijkstra(G, w, s)
2   for each vertex v in V[G] // Initializations
3     d[v] := infinity
4     previous[v] := undefined
5   d[s] := 0
6   S := empty set
7   Q := set of all vertices
8   while Q is not an empty set // The algorithm itself
9     u := Extract_Min(Q)
10    S := S union {u}
11    for each edge (u,v) outgoing from u
12      if d[v] > d[u] + w(u,v) // Relax (u,v)
13        d[v] := d[u] + w(u,v)
14        previous[v] := u
  
```

We are not going to explain all the Dijkstra algorithm again, we will add just some hints to understand how it works in the implementation.

This is an iterative algorithm, each iteration it will repeat the same. Firstly, it will select the closest node. With the closest node we will evaluate all the possible outgoing edges from this node and we will do the relax method to recalculate the costs and predecessor from the opposite node.

After Dijkstra's execution the result will be the following one:



Get optimal path

Now, the graph is ready but we do not have the final result yet. Looking at the last figure, it's advisable that the cost to arrive to each node is the optimal for that node, so if we come back using its predecessors, we will get the whole path.

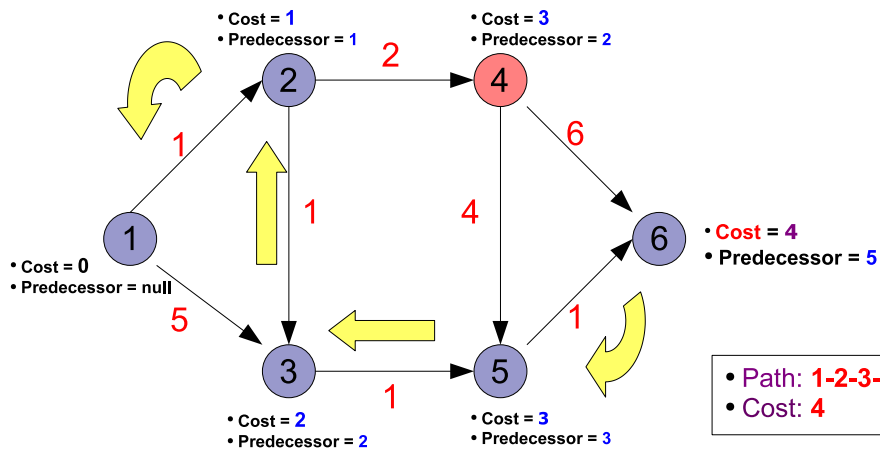
PSEUDO-CODE

```

1 S := empty sequence
2 u := t
3 u.cost := cost of the whole path
4 while defined u
5     insert u to the beginning of S
6     u := previous[u]
7 s := final path

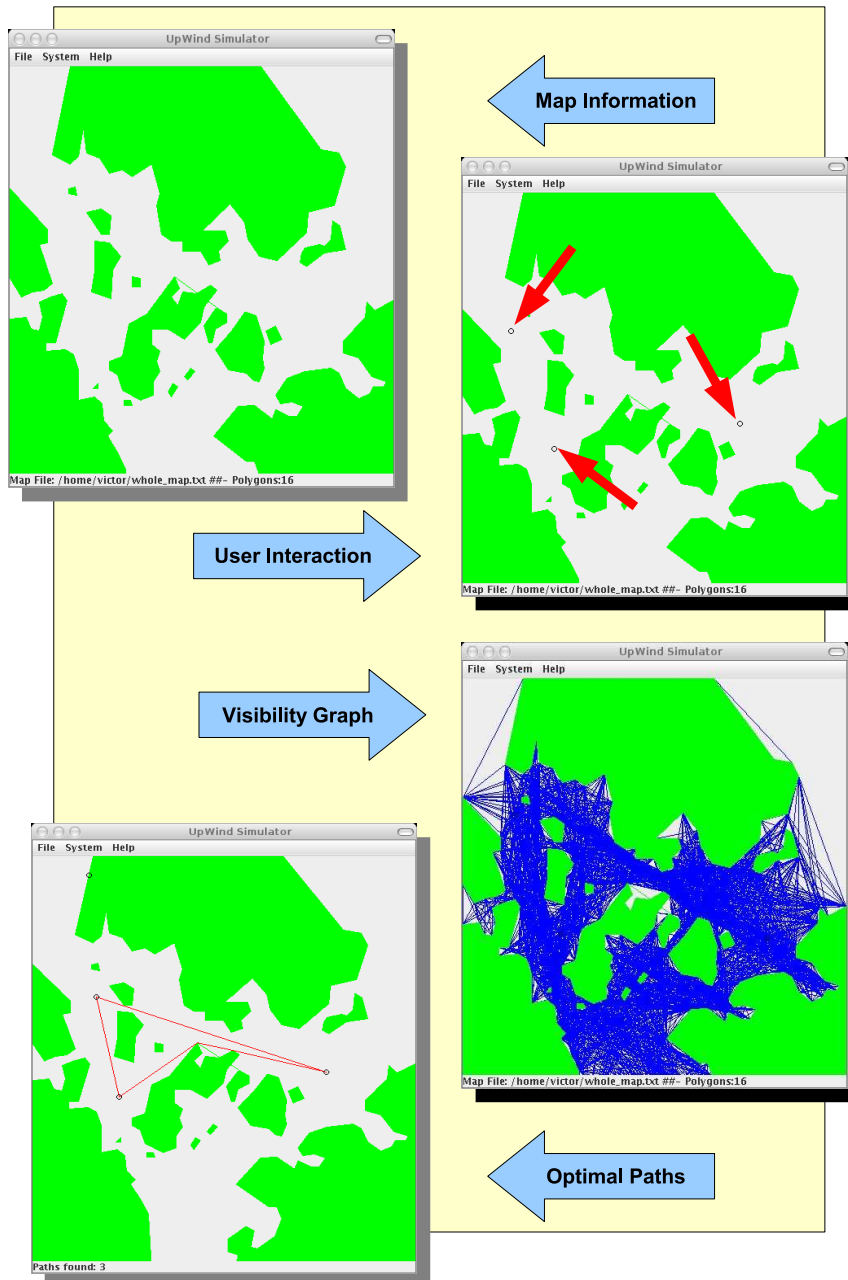
```

Then, the last step that we have to do to get the final path is just choose the node to arrive to and come back using its predecessors. In the example underneath we will come back from the last node of the graph.



4.3.7 Result: Optimal path

After all this work is the time to show the yearned result. The following application screen shots will show the successful result of this project.



As is possible to see in the above figure, the final goal of this project has been achieved. The four screen shots show the steps to get the final result.

Load Map Info → Add control points → Visibility Graph → Dijkstra → Final Result

Project final evaluation

Contents

5.1	Achievements	58
5.2	Next steps	59

5.1 Achievements

We would like to gather all the main goals we have managed to get in this project. These are the following ones:

- I/ ***Project Definition:*** Before this project had begun, there were some ideas about what we wanted to get finally. As we said, they were just ideas that with the development and thanks to the work done in this project they have become real facts.
- II/ ***UpWind Simulator:*** We think that the UpWind Simulator java application developed for this project is a very good base to implement further improvements and become a final working application.
- III/ ***Simple User Interface:*** We have managed to get the most useful application without useless features, sometimes is important to keep it simple in order to achieve the best result.
- IV/ ***Solid Base:*** Well-known mathematical theorems have been used to get the final simulator prototype. We, as computer engineers, have researched, analysed, implemented and used the algorithms that would suit in the context our project.
- V/ ***Visibility Graph design:*** The main requirement in order to achieve the best result of Dijkstra's algorithm was to build the perfect visibility graph and build it without errors. Finally we succeeded it.
- VI/ ***Dijkstra's Algorithm implementation:*** A correct implementation of Dijkstra's algorithm has been done and the result is the best proof.

5.2 Next steps

We think that would be quite useful for future UpWind members to know how to go on with this project. We would like to write the things we would do next like possible goals to get.

- I/ ***ESRI file integration:*** One of the most important things is to work with real maps information. With real data would be possible to do real test of the simulator. We have defined our own file format because integrate ESRI wasn't our task at all but would be really good if our map file format could be replaced by real ESRI files.
- II/ ***Delimit Visibility Graph:*** At the moment, when the map file is opened, the visibility graph is created taking into account all the islands or polygons of the map. We are working with small maps so the visibility graphs are not too big and the calculations are quite fast. However, this will be a problem if we try to load big maps. Somehow, it should be possible to delimit the area to build the visibility graph.
- III/ ***Dijkstra's cost function:*** As we commented in some point in this documentation, we are calculating the optimal paths using the Euclidean distance equation. This is not the final goal of the simulator, we are just taking island information into account and it would be really useful if the cost method were improved adding more and more factors.
- IV/ ***GPS data handling:*** At some point of this project, dynamic information will be needed. This information could be linked with wind direction, wind speed, ship speed, other boats location, etc. . . This information will be provided by a GPS receiver, so the way of handling the information of this GPS receiver will be another point to develop.

Bibliography

- [1] Franco P. Preparata, Michael Ian Shamos.
"Computational Geometry an introduction."
Springer-Verlag, 1988.
- [2] Joseph O'Rourke.
"Computational Geometry in C."
Cambridge University Press, 1998.
- [3] M. de Berg, M. van Kreveld, M. Overmars, O. Schwarzkopf.
"Computational Geometry, algorithms and applications."
Springer-Verlag, 2000.
- [4] Xerox Palo Alto Research Center. *"Graphic Gems"*
Morgan Kaufmann, 1990.
- [5] Program of Computer Graphics Cornell University. *"Graphic Gems II"*
Morgan Kaufmann, 1991.
- [6] California Institute of Technology Computer Graphics Laboratory. *"Graphic Gems III"*
Morgan Kaufmann, 1992.

Appendix

Contents

6.1	UpWind Simulator class overview	64
6.1.1	Simulator.java	64
6.1.2	Map.java	64
6.1.3	CVisibilityGraph.java	65
6.1.4	Dijkstra.java	65
6.1.5	Path.java	65
6.1.6	CPolygon.java / CVertex.java / CEdge.java	65
6.2	Resources used	66
6.3	Project Logo	67
6.4	Acronyms	68

6.1 UpWind Simulator class overview

The project is composed by the following class files:

- Simulator.java
- Map.java
- CVisibilityGraph.java
- Dijkstra.java
- Path.java
- CPolygon.java
- CVertex.java
- CEdge.java

We will describe them slightly in order to help the future developers.

6.1.1 Simulator.java

In this class user interface is defined. We have defined all the methods to create the interface and to interact with it like button and mouse events. We have separated clearly the user interface from the rest of the application.

6.1.2 Map.java

All the issues linked with the map handling are implemented in this class: file map opening, file map drawing, etc. . . . The visibility graph and Dijkstra's execution will be executed from this class.

6.1.3 CVisibilityGraph.java

This is the class where all the main calculations are implemented. The input handling, input transformation, visibility graph force brute construction and Dijkstra execution are placed in this class.

6.1.4 Dijkstra.java

The main methods to execute Dijkstra's algorithm are implemented here.

6.1.5 Path.java

The path class defines the path object used to handle easily the last results in Dijkstra's algorithm-

6.1.6 CPolygon.java / CVertex.java / CEdge.java

These are the objects used to store and handle the information about islands. They do possible to build the visibility graph and Dijkstra's algorithm.

6.2 Resources used

Next we are going to list all the resources used in this project:

- Eclipse SDK 3.1.2: <http://www.eclipse.org>
- Eclipse plugin Visual Editor: <http://www.eclipseplugincentral.com>
- Latex used to create this documentation: <http://www.latex-project.org>
- JSmooth tool used to wrap java jar files and create executable files for windows [.exe]: <http://jsmooth.sourceforge.net>
- Scribus used to create the presentations: <http://www.scribus.net>
- Open Office Draw used to create the documentation figures: <http://www.openoffice.org>

Note that all the project has been developed using open source tools in order to achieve the open source OSI certification when the project is finished.

http://www.opensource.org/docs/certification_mark.php



6.3 Project Logo

The UpWind project logo has been designed by *Oscar Valls Miralles*.

He is a Spanish graphic designer and like collaborator in this project, the UpWind members would like to give thanks to him for spending his free hours designing some sketches to get finally the logo we liked most.



6.4 Acronyms

GPS ▶ Global Positioning System

OSPF ▶ Open Shortest Path First

OSI ▶ Open Source Initiative

SDK ▶ Software Development Kit

ESRI ▶ Environmental Systems Research Institute (developer of geographic information systems (GIS) software, Redlands, California)