Université de Pierre et Marie Curie

Course Project 'Initiation au C++'

Fast search of a triangle covering a point in a mesh

Arne Heimendahl, Olivia Kaufmann

Supervisor: F. Hecht, X. Claeys

Submission Date: 11/01/2018

Contents

1 Project description and aim

The aim of this project is to implement an algorithm which searches for a triangle in a convexe mesh covering a point $p = (x, y)$. A linear complexity in number of vertices of the mesh is desired.
A mesh is a set of triangles where the intersection of two triangles is either empty, a vertex or an edge. See Figure 1 for an illustration.

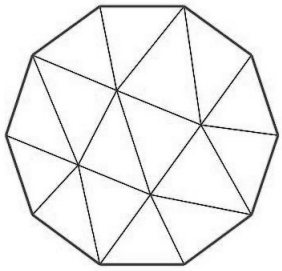


Figure 1: Example for a mesh

1. The task is to implement a class `Mesh` who stores arrays of vertices and triangles. The data shall be imported from a mesh file.
2. A function given an input triangle and one of its vertices returning an adjacent triangle opposite to the vertex shall be implemented. The complexity should be constant after an unique initialization of the neighbors in $O(n_T)$ or $O(n_T \log(n_T))$.
3. An algorithm starting in a given triangle and walking until finding the covering triangle of a specified point is demanded.
4. Thereafter, the results shall be displayed.
5. Given two meshes, `m` and `M`, covering triangles in `M` of the vertex set of `m` shall be searched.

2 The template class T3 and the class Triangle

2.1 The template class T3

Objects of the class `T3` represent elements of a three dimensional space in `T`. Hereby, the data type `T` is defined using a template. Within this project, either `T3<int>` for the definition of a triangle (see 2.2) or `T3<double>` for the definition of the coordinates of a vertex (see 3) is used. The private members x, y and z of type `T` store the entries of the `T3` vector. The class has a default constructor, a constructor by copy and a constructor creating a vector given the three vector entries. Moreover, the class has operators to access and modify the entries, add to elements of `T3`, multiply an element of `T3` by a scalar of the type `T` and calculate the scalar product of two `T3` vectors. The operator $<$ compares two elements of `T3`, $v_1 = \begin{pmatrix} x_1 & y_1 & z_1 \end{pmatrix}^T$ and $v_2 = \begin{pmatrix} x_2 & y_2 & z_2 \end{pmatrix}^T$, in the following way:

$$v_1 < v_2 \quad \Leftrightarrow \quad \begin{cases} x_1 < x_2 \text{ or} \\ x_1 = x_2 \text{ and } y_1 < y_2 \text{ or} \\ x_1 = x_2 \text{ and } y_1 = y_2 \text{ and } z_1 < z_2 \end{cases} .$$

This allows a lexicographic ordering of a list of `T3` elements (see 3.2.2, `T3<int>`). For three `T3` vectors, which define the vertices of a triangle, the method `T3::oriented_vol` computes the corresponding signed volume. Let a, b and c denote `T3` vectors. The *oriented volume* is defined as

$$\left(\overrightarrow{ab} \times \overrightarrow{bc}\right)^T \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} = (b_1 - a_1)(c_2 - b_2) - (b_2 - a_2)(c_1 - b_1)$$

The sign is positive for triangles oriented in trigonometric sense. This method is used in the method `Mesh::Promenade` 3.3 to determine a suitable 'walking' direction.

2.2 The class Triangle

The class `Triangle` inherits form the class `T3`. Its derived members i, j, k are specialized as integers representing the position of its defining vertices in the array of the points of the given mesh. This array is a member of the class `Mesh` in 3. In addition, it has three extra integer members `Triangle::int neighbor1, neighbor2, neighbor3` representing the position of the adjacent triangles in the array of triangles which is also a member of the class `Mesh`. Notice that theses index lies between zero and the number of triangles minus one.
An adjacent triangle $t^{'}$ is said to be `neighbor1` of a triangle t if, saying i represents the first index of a vertex of t but i does not represent a vertex of $t^{'}$. The same holds for `neighbor1` and `neighbor2`. Note that this relation is not symmetric, that is to say that $t^{'}$ is `neighbor1` does not imply that t is `neighbor1` of $t^{'}$. This assignment facilitates the access to the following adjacent triangle in the algorithm `Promenade` (see 3.3).
At the creation of a new triangle the latter three members are initialized by -1 which means that a triangle has no neighbors when it is created. The functions `Mesh::setAdjacencyViaMultimap` (see 3.2.1) and `Mesh::setAdjacencyViaList` (see 3.2.2) set the neighbors via the stl containers multimap, respectively list. After the execution of one of these two functions `neighbor1` $= -1$ describes the case that there is no adjacent triangle on the opposite side of the first vertex. Since the neighbors are private members there are getter and setter in order to read and manipulate them.

3 The class Mesh

The class `Mesh` contains all necessary information of the mesh and the search of a vertex in (or outside) the mesh can be realized by its member functions. Its private members are pointers to the arrays of vertices and triangles of the mesh as integers who store the size of these two lists. In order to create an object of type `Mesh` the name of the desired .msh file has to be transmitted. The file is read by the functions `Mesh::LoadVertices` and `Mesh::LoadTriangles` 3.1 who then initialize the members `Mesh:: Triangle* triangles, T3<double> vertices, int numbTriangles, int numbVertices`. Since the members are private there are getter and setter for `numbTriangles` and `numbVertices` as well as getter for `triangles` and `vertices`. The functions `LoadTriangles` and `LoadVertices` are their setters.

3.1 LoadVertices and LoadTriangles

Both functions are called in the constructor of the class `Mesh` setting the members `vertices` and `triangles`. They work basically the same way with the small difference that they create arrays of different data types (`T3<double>` and `T3<int>`) and search for different key words in the .msh file. At first, the .msh file is opened according to its name. Then the functions search for the line indicating the number of vertices respectively the number of triangles. This is implemented by using an `std::fstream` and the function `std::getline` who stores the information of a line and skips to the next line.
To make the code work the next line must include the according number which is then stored in `numbVertices`, respectively `numbTri`. The data is read as a string which is transformed to an integer by the command `astoi`. These numbers also define the size of the arrays of type `T3<double>` and `Triangle` which are created by `new` in order to allocate the necessary memory. The following lines must include the coordinates of all vertices, respectively the positions of the triangles in the table of vertices. The lines are read as strings who are casted in to doubles, respectively integers.

Each line defines a vertex or a triangle which is then stores in the correspondent array. Both functions finally return pointers to the arrays filled with the vertices or triangles.

3.2 Find the adjacent triangle

3.2.1 setAdjacencyViaMultimap

The goal of this function is the initialization of the members `neighbor1, neighbor2, neighbor3` of the class `Triangle`. As a reminder, they are defined by their position in the array of triangles. Their initialization is realized by the container `std::multimap`. Each triangle $t = (v_{i_1}, v_{i_2}, v_{i_3})$ is represented by the sequence (i_1, i_2, i_3), where these indices indicate the position of the vertices v_{i_1}, v_{i_2} and v_{i_3} in the array of vertices. For each t three pairs are added to the multimap, where the edges (v_{i_k}, v_{i_l}), $k, l \in \{1, 2, 3\}$, $k \neq l$, stored by the data type `std::pair<int,int>` (more precisely `pair<i_k,i_l>`), represent the keys, whereas the mapped value is an integer representing the position of the triangle in the array of triangles. Hence, the initialization if the multimap is realized in $3n_T$ loop runs, so in $O(n_T)$.
Essentially for the functioning of the method is the ordering of the of the indices i_1, i_2, i_3 in t. As in `setAdjacencyViaList`, an edge $\{v_{i_k}, v_{i_l}\}$ has to be uniquely identified by the pair (i_k, i_l) which should not be mixed up with (i_l, i_k). This is achieved by demanding that for each triangle $t = (v_{i_1}, v_{i_2}, v_{i_3})$ the vertices are ordered in a manner such that $i_1 < i_2 < i_3$. Thus, an edge (v_{i_k}, v_{i_l}) can uniquely be identified by the pair (i_k, i_l), $i_k < i_l$.
We find the adjacent triangles by the following three steps:

1. We range over the multimap and in each step we save the current key (i_k, i_l), its position in the list (this is the current iterator) and the index i of the triangle $t = (v_{i_1}, v_{i_2}, v_{i_3})$. Then we move the iterator to the next element of the list and erase the pair $((i_k, i_l), i)$ of the multimap which is realized in constant time since the iterator belonging to $((i_k, i_l), i)$ was saved.

2. Now searching by the function `find` for the stored key (i_k, i_l) allows us to determine if there is another index j associated to (i_k, i_l). If this is the case (the iterator returned by find does not point to the last element of the multimap) the triangles represented by i and j are adjacent. The function $find$ needs $O(\log(n_T))$ in order to return an iterator. Note that the pair $((i_k, i_l), j)$ is not erased. This would yield an error because the actual iterator would try to access that element.

3. The indices i and j are set as the neighbors of each other where j is set as neighbor $k \in \{1, 2, 3\}$ if the index k does not represent a vertex of the triangle represented by j. The same rule is applied to i as a neighbor of j. This assignment is realized in constant time.

The application of all three steps needs a time of $O(n_T)O(\log(n_T)) = O(n \log(n_T))$.

3.2.2 setAdjacencyViaList

As in `setAdjacencyViaMultimap`, this method sets the members, describing the neighbors, for each triangle in the considered mesh. In the same manner as before, the members are described using their position in the triangle array. The idea of this method is to create a list in which adjacent triangles appear in consecutive order. After this list is created, the adjacencies of the triangles can be set in linear complexity in the number of triangles.
The list with the desired property is created in the following way: For each triangle three triples `T3<int>` are created and appended to the list. For the triangle $t = (v_{i_1}, v_{i_2}, v_{i_3})$ with the index i, a new `T3<int>` for every edge is defined. More precisely, the triples $(i_1, i_2, i), (i_1, i_3, i)$ and (i_2, i_3, i) are pushed on the list. This can be done in $O(n_T)$. As in `setAdjacencyViaMultimap`, it is necessary that an edge is with increasing vertex indexing, so $i_1 < i_2 < i_3$ has to hold. In order to archive that two neighboring triangles are positioned one after another, the list is sorted in lexicographic order (see 2.1). First, edges of triangles

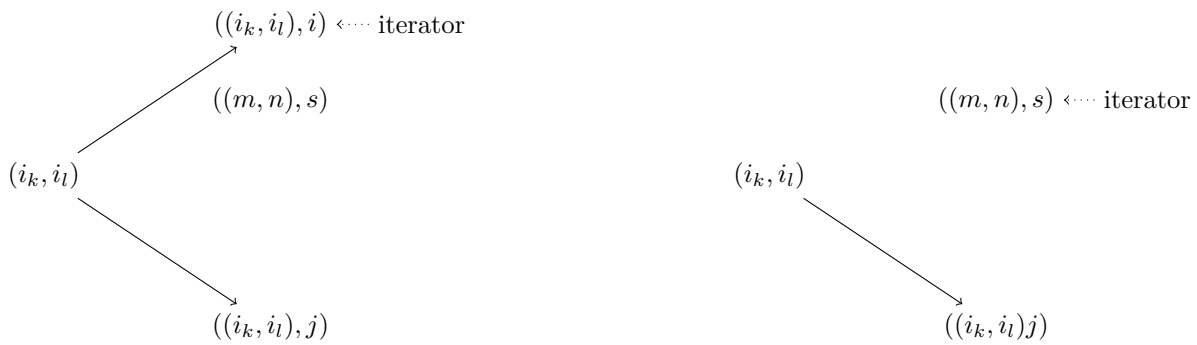


Figure 2: The initial situation for determining the neighbors (left). The element$((i_k, i_l), i)$ is erased from the map in the iterator points to another element $(((m, n), s)$ in this case), not necessarily $((i_k, i_l), j)$. This essentially depends on the internal ordering of the multimap.

including vertex 1 are listed. These are ordered with respect to the second triple entry. For two list entries with coinciding edges, the list is ordered with respect to the triangle index stored in the last triple entry. In this case, two neighboring triangles are considered. Then, triples with leading entry 2 followed by higher indices that are not yet considered are appended. In Figure 3 the structure of the sorted list is depicted.

$$
\begin{array}{ccc}
1 & * & * \\
1 & * & * \\
 & \vdots & \\
i_1 & i_2 & i \\
i_1 & i_2 & j \\
i_1 & i_3 & i \\
 & \vdots & \\
i_1 & * & * \\
i_2 & i_3 & i \\
i_2 & i_3 & k \\
 & \vdots & \\
n_T & * & *
\end{array}
$$

Figure 3: Illustration of the list structure

The increasing order for indices of an edge is crucial since otherwise neighboring triangles would not appear in consecutive order. The complexity for sorting this list of size $3n_T$ is $O(n_T log n_T)$.
After the creation of this list, the members of the mesh triangles can be set. Due to the lexicographical ordering, the list allows now to define adjacencies between triangles with a number of comparison which is linear in the number of triangles. Two consecutive triples of the list are considered. The first two entries are compared. If they are equal, the corresponding triangles share an edge and, hence, can be defined as adjacent. Particularly, if the triangles with the indices i and j have the vertices v_k and v_l in common, the triples (k, l, i) and (k, l, j) appear consecutively in the list. The head element of the list is popped from the front and the comparison is repeated for new top triples. If the first two entries of successive triples do not coincide, the neighbor attached to the current edge was either already set in the step before, or

the edge is located on the boundary of the considered mesh. In this case, the corresponding triangle's member is not modified and will still be defined as -1. The comparison of every pair of triples has constant complexity. As a list of size $3n_T$ is considered, the complexity of setting the neighbors given this list is $O(n_T)$.
In summary, the method `setAdjacencyViaList` has complexity $O(n_T log n_T) + O(n_T) = O(n_T log n_T)$.

3.2.3 Runtime comparison of setAdjacencyViaMultimap and setAdjacencyViaList

The complexity for both methods, `setAdjacencyViaMultimap` and `setAdjacencyViaList`, lies in $O(n_T log n_T)$. Measuring the average runtime for setting the neighbor for the 21870 vertices of mesh5.msh leads to the following results:

- `setAdjacencyViaMultimap`: $0.0887504s$
- `setAdjacencyViaList`: $0.0750262s$

Observe that we only have minimal difference in the runtime for both methods. The measurements are proceeded in the file TimeMeasurements.cpp.

3.3 The algorithm Promenade

`Mesh::Promenade` is a method of the class `Mesh` with input variables triangle t and a point p of the type `T3<double>`. As output a triangle in the mesh covering p is returned.
Let $t = (c_1, c_2, c_3)$, where the vertices c_1, c_2 and c_3 are of type `T3<double>`. Hereby, the triangle is considered in trigonometric sense. Thus, the oriented volume of the triangle t is positive. Recall that `oriented_volume` is a method of the class `T3` which takes two additional `T3` vectors as input (see 2.1).
For each edge of the triangle t, a new triangle containing the edge and the point p is considered. We are interested in the oriented volumes of the triangles $(c_1, c_2, p), (c_2, c_3, p)$ and (c_3, c_1, p). If the oriented volume of one of those new triangles is positive, then p and the triangle t are contained in the half plane defined by the currently considered edge of t. Consequently, p is contained in t if the oriented volumes of the triangles $(c_1, c_2, p), (c_2, c_3, p)$ and (c_3, c_1, p) are all positive. In this case, the method `Promenade` returns the input triangle t. On the other hand, if there is one triangle with negative oriented volume, t is not covering p. Hence, t and p are separated by the regarded edge. We choose one edge e who leads to a negative oriented volume for the triangle formed with p. This neighbor can be chosen via `random_neg` (see 3.3.1) or `min_neg` (see 3.3.2). Then, we consider the adjacent triangle n sharing the edge e with t. Thereafter, the method `Promenade` is called recursively with starting triangle n. With this choice of neighboring triangles, the path generated by the algorithm heads in the direction of p. In addition, it should be paid attention to the existence of a neighboring triangle. Triangles on the boundary of the mesh just have two neighbors, the remaining member is initialized by -1.
In order to make sure that a triangle is treated in trigonometric sense, the oriented volume of the starting triangle $t = (c_1, c_2, c_3)$ is consulted. If `oriented_volume` returns a negative value for the oriented volume, the results for oriented volumes of the triangles $(c_1, c_2, p), (c_2, c_3, p)$ and (c_3, c_1, p) are negated. Crossing an edge from one triangle to an adjacent one, the sense in which this edge is viewed has to be reversed to guarantee a consideration of the triangles in trigonometric sense. Using the above approach, this can be treated efficiently without need to know the traversed edge and its indices in the neighbor triangle.
Optionally, the method `Promenade` takes also a vector with entries of the class `Triangle`, called `path`, as input variable. The current triangle is pushed back on the vector and, due to recursively calling of the method, the vector stores the consecutively passed triangles when the algorithm terminates. The path is used to visualize the results of the method `Promenade` in 3.4.
The method terminates when the desired covering triangle, if it exists, is found. Otherwise, if the point p is located outside of the mesh, the method returns a triangle on the boundary of the mesh. The only

edge forming a triangle with negative oriented volume with p will be part of the mesh boundary. Thus, no adjacent neighbor exists. The returned triangle is the closest triangle to p.

Following the algorithm, there are three cases for the number of possible walking directions:

1. Exactly one edge forms a triangle with negative oriented volume with p,
2. two neighboring triangles are valid choices or
3. the current triangle already covers p.

If there are two neighboring triangles to choose from, different approaches can be pursued. We implemented a random neighbor selection (see 3.3.1) and a method choosing the edge forming the triangle with minimal oriented volume with p (see 3.3.2). In order to switch between the two approaches, adapt the switch-condition in the method `Promenade` in line 128 and 160 in Mesh.hpp `Mesh.hpp` . XXXXXXXXXXXXX

3.3.1 Random neighbor selection

The method `random_neg` is given three numbers of the type `double` and returns an integer. If at least one input variable is strictly negative, the output takes a value between one and three indicating the handover position of a negative input variable. Otherwise, if non of the input variables is negative, -1 is returned. In the case of several negative input variables, the variable is chosen randomly. Hereby, the command `srand(time(NULL))` is used to generate random numbers.
Using this approach for the choice of the neighbor triangle in `Promenade`, the algorithms returns different paths when executing it repeatedly with the same starting triangle and point p.

3.3.2 Selection of neighbor with minimal oriented volume

Another option to choose the consecutive triangle is to select the edge forming the triangle with minimal oriented volume with the point to cover. The method `min_neg` returns the smallest of the three given numbers or -1 if none of them is negative.
Unlike `random_neg`, the neighbor choice with `min_neg` is deterministic. So the returned path is unique, unless two edges and p form triangles with the same negative oriented volume. As explained in the following sections, the neighbor selection with minimal oriented volume turns out to accelerate the algorithm `Promenade`.

In Figure 5 the path of `Promenade` returned for a specified starting triangle and point to cover are depicted. Hereby, the first path is generated using `min_neg` whereas the other three ones are created using `random_neg`. As expected, the path return when `min_neg` is used, is straighter and shorter than the pathes generated using `random_neg`.

3.3.3 Can an arbitrary choice of the consecutive triangle be a better choice?

This example shows that a random choice of the consecutive triangle can be better.
Assume that the algorithm `promenade` currently considers the triangle A of Figure 4 and searches for a covering triangle of the point p. Hence, the algorithm will proceed with triangle B or C.
If we use `min_neg`, we compare the oriented volume of the convex hull of p and the lower right edge of A and the convex hull of p and the upper right edge of A. We will clearly choose C, as illustrated in the figure on the right as the green area is clearly bigger than the blue one. This is obviously the worse choice since p as a vertex of B is already covered by B.
An important observation is that the regularity of the triangles plays an important role. In this case,

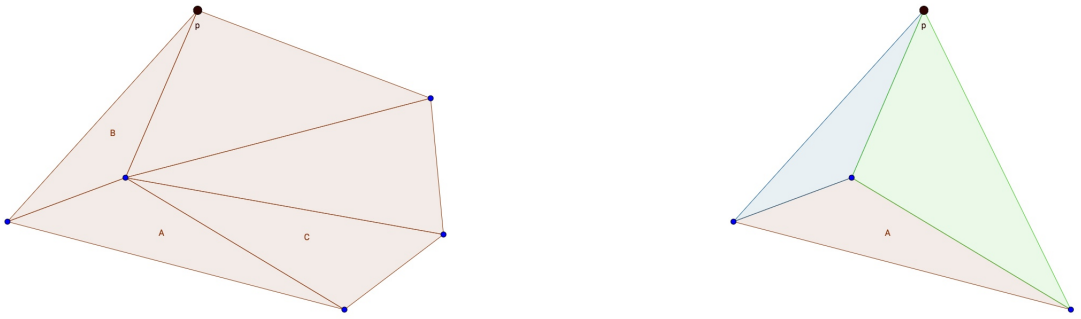


Figure 4: The illustration of 3.3.3.

regularity means that all triangles of the mesh have a 'similar' area, 'similar' angles and its vertices are adjacent to a 'similar' number of triangles.

3.3.4 Empirical demonstration that the deterministic choice of the consecutive triangle is better in regular meshs

If we take a look at the figures 5 and 6, it becomes empirically clear that the minimum choice of the consecutive triangle speeds up the algorithm. Figure 5 even states that the path created by the minimum choice is always shorter or as long as the path created by the random choice. Looking at the runtime in 6, we see that the runtime is about as twice as high for the random choice when the starting triangle and the given point are remote.
Nevertheless, it has to be kept in mind that the underlying mesh plays an important role. In this case, the mesh 'mesh5.msh', used for the figures 5 and 6, is regular in the sense explained in the previous section (see 3.3.3). These `gnuplot` figures are created in the file TimeMeasurements.cpp.

3.4 The visualization via gnuplot

The visualization is done by the function `Mesh::exportGnuplot` which is able to visualize two different inputs:

1. The result of the algorithm Promenade, that is a sequence of adjacent triangles from a starting triangle to a triangle covering the searched point. In this case, the input variable `vector<triangle> triangles_path` contains the sequence of triangles and the input variable `const T3<double>* points` contains just one point, hence, the input `int numbpoint`, denoting the size of the array `points`, is 1.

2. Given a mesh and the vertices of another mesh, it can visualize the set of the covering triangles (a subset of triangles of the first mesh) of the points of the second mesh. In this case, the input variable `vector<triangle> triangles_path` contains the covering triangles and `points` stores the vertices of the second mesh.

The function writes four text files, one for all triangles of the mesh, one for the triangles in `triangles_path`, one for the points in `points` and one for the commands which shall be executed by `gnuplot`. This is all realized by a simple `std::ofstream` variable, allowing to create and manipulate a text file.
The script for `gnuplot` contains a line which keeps the plot open until some key is hit in the terminal. The actual execution in the terminal is achieved by the command `system` which executes an input string in the terminal in order to call `gnuplot`.

4 Cover vertices of a mesh by another mesh

The aim of this section is to find the covering triangle for every vertex of another mesh in the same domain. For this purpose, the Promenade algorithm is executed many times, once for every vertex. In order to still obtain a good runtime, the starting triangle for `Promenade` should be chosen in an efficient way. The idea is to use starting triangles for which we already know that they are located close to the vertex to cover.
The function `findVertices` has two meshes, `Mesh m` and `Mesh M`, and a vector of triangles called `coveringTriangles` as input variables and returns these `coveringTriangle` after modification. The function's goal is to search for covering triangles in `M` for the vertices of `m`. After execution of `findVertices` the i-th array entry of `coveringTriangles` stores the triangle in `M` which covers the i-th vertex of the mesh `m`. All covering triangles are initialized by 0. Note that the indexing of the vertices in the mesh files starts at 1.
First, two random triangles, one in `m` and one in `M`, are chosen. In `findVertices` covering triangles for the vertices of the randomly chosen triangle `firstTriangle_m = (firstVertex, secondVertex, thirdVertex)` in `m` are computed. In order to find a covering triangle for `firstVertex`, `Promenade` is executed given the random triangle `firstStartTri` chosen in the `M` as input. After calculation of the covering triangle for `firstVertex`, indexed by `firstTriangle_M`, we can use that triangle as the starting triangle for the search of a covering triangle for `secondVertex`. As `firstVertex` and `secondVertex` belong to one edge in `m`, we expect the covering triangles of those vertices to be close in `M`. In the same manner, we use the covering triangle of `secondVertex` as starting triangle for covering `thirdVertex`. Hereby, the version of `Promenade` not storing the taken path is executed.
Now, we computed covering triangles for all the vertices of `firstTriangle_m`. Originating from this triangle, we start a recurrence to find the remaining covering triangles. Neighboring triangles, neighbors of the neighboring triangles and so on will be considered.
`findVertices` calls the function `Triangle_Recurrence`. The two meshes, the vector `coveringTriangles` and the already covered triangle `firstTriangle_m` are passed. `Triangle_Recurrence` modifies the vector `coveringTriangles` and, thereafter, `findVertices` returns the completed vector. `Triangle_Recurrence` searches for covering triangles for the vertices of neighboring triangles of the already covered triangle. Again, 'close' covering triangles that are already found are used to accelerate the termination of the algorithm `Promenade`.
We start with the member `neighbor1` of `firstTriangle_m`. If this neighbor triangle exists and its vertices are not yet covered, we do so using the same procedure as in `findVertices`. After the triangle is covered, we consider its member `neighbor1`. At the moment when either a neighbor does not exist or is already covered, we go on to the member `neighbor2` and afterwards to the member `neighbor3`. If all members are covered, we return to the calling triangle and considered its next neighbor. This terminates when a covering triangle for all vertices of the mesh `m` is found and, consequently, the vector `coveringTriangles` is completed. Observe that this searching structure is similar to the approach of depth-first search.

5 Test environment - the main function

The file main.cpp offers the possibility to test the properties of the program:

1. Find the neighbors of a triangle. If a valid triangle index is chosen, the triangles and its neighbors are marked in the chosen mesh. This case offers the extra option to choose if the adjacency should be set via list or multimap. Essentially, the functions `setAdjacencyViaList` or `setAdjacencyViaMultimap` and `exportGnuplot` are called.

2. Find the covering triangle for a selectable point. As a result, the path to the covering triangle of the point is displayed. If the point lies outside of the mesh, the path to the nearest triangle is marked

in the mesh. In this case, the functions `Promenade` and `exportGnuplot` are called.

3. Find the covering triangles for the vertices of another mesh. These option offers to display all covering triangles of a the vertices of a second mesh. Note that the mesh whose vertices is supposed to be coarser (a finer mesh can be embedded in a coarser mesh as well but then each triangle serves as a covering triangle). By covering the vertices of a coarser mesh with the finer mesh, we can guarantee that all vertices can actually be covered by triangles of the finer mesh. This holds since the domain of the coarser mesh is included in the domain of the finer mesh if the domain defining intervals coincide.

All three options allow to choose the mesh where the mesh becomes increasingly finer from mesh1.msh to mesh5.msh. Of course, it is also possible to add own meshes. This can be done with `Gmsh`. Afterwards, the mesh files exported from `Gmsh` have to be prepared in order to provide the needed file structure and the ascending order of the vertices' indices within a triangle. This can be done with prepareMeshFile.cpp. Moreover, the mesh's interval should be at maximum $[1.5, 1.5]$ for a convenient display since the axis of the plot for `gnuplot` are predefined as the interval $[2, 2]$.
The test environment is not robust face to wrong inputs. That means that, for example, if an integer is demanded but a letter is typed, the program has to be started again. The same happens if an index is demanded but a the typed input lies outside of the range (an assert exception is thrown then).
The program main.cpp can be launched by typing the command `make do` in the terminal (the `Makefile` is called). By typing `make clean` the current .txt and .o files are erased.

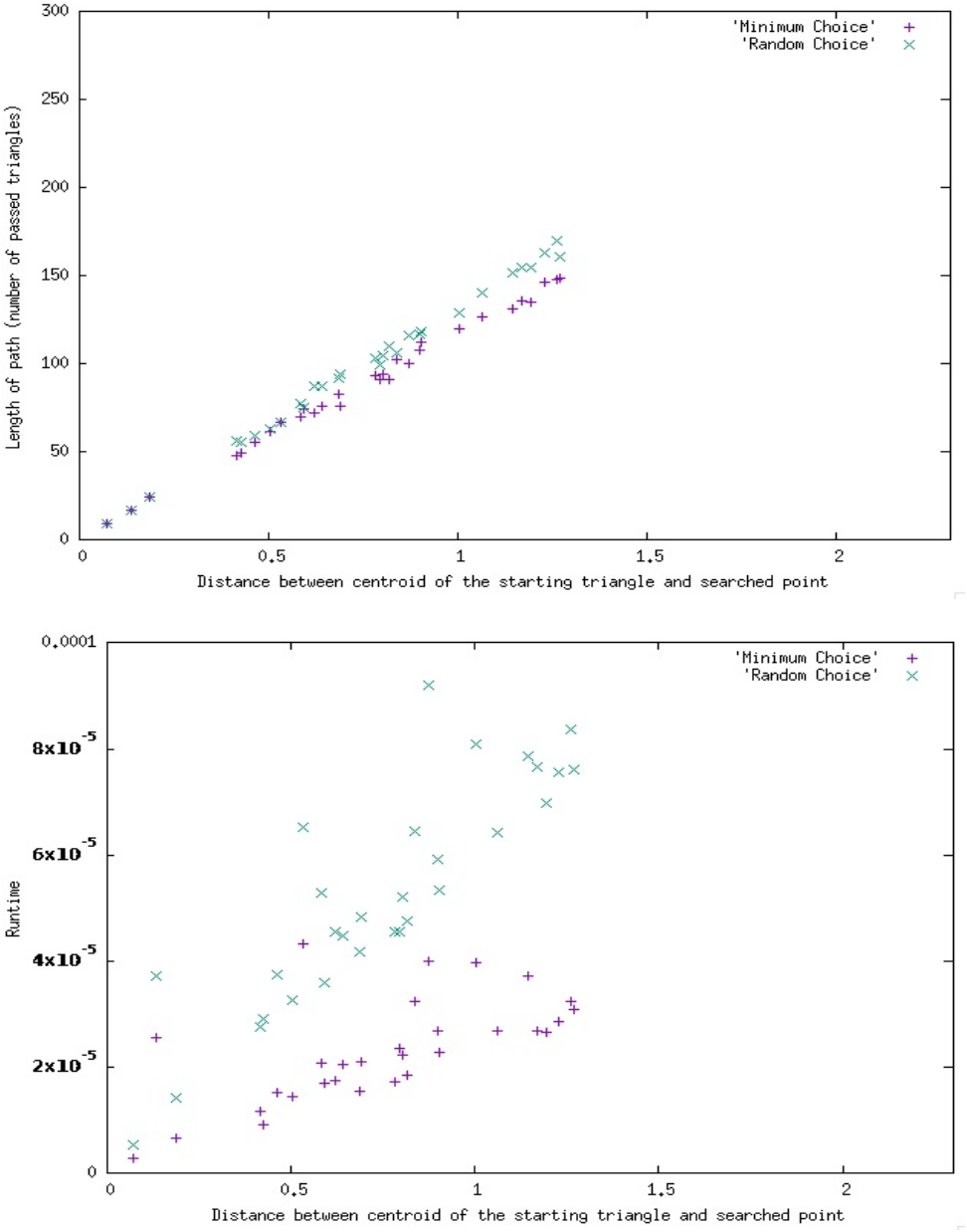


Figure 5: The pathlength and the runtime compared using min_neg and $random_neg$. In this example, 30 data points were randomly created in the file 'TimeMeasurements.cpp'. The underlying mesh is 'mesh5.msh'.

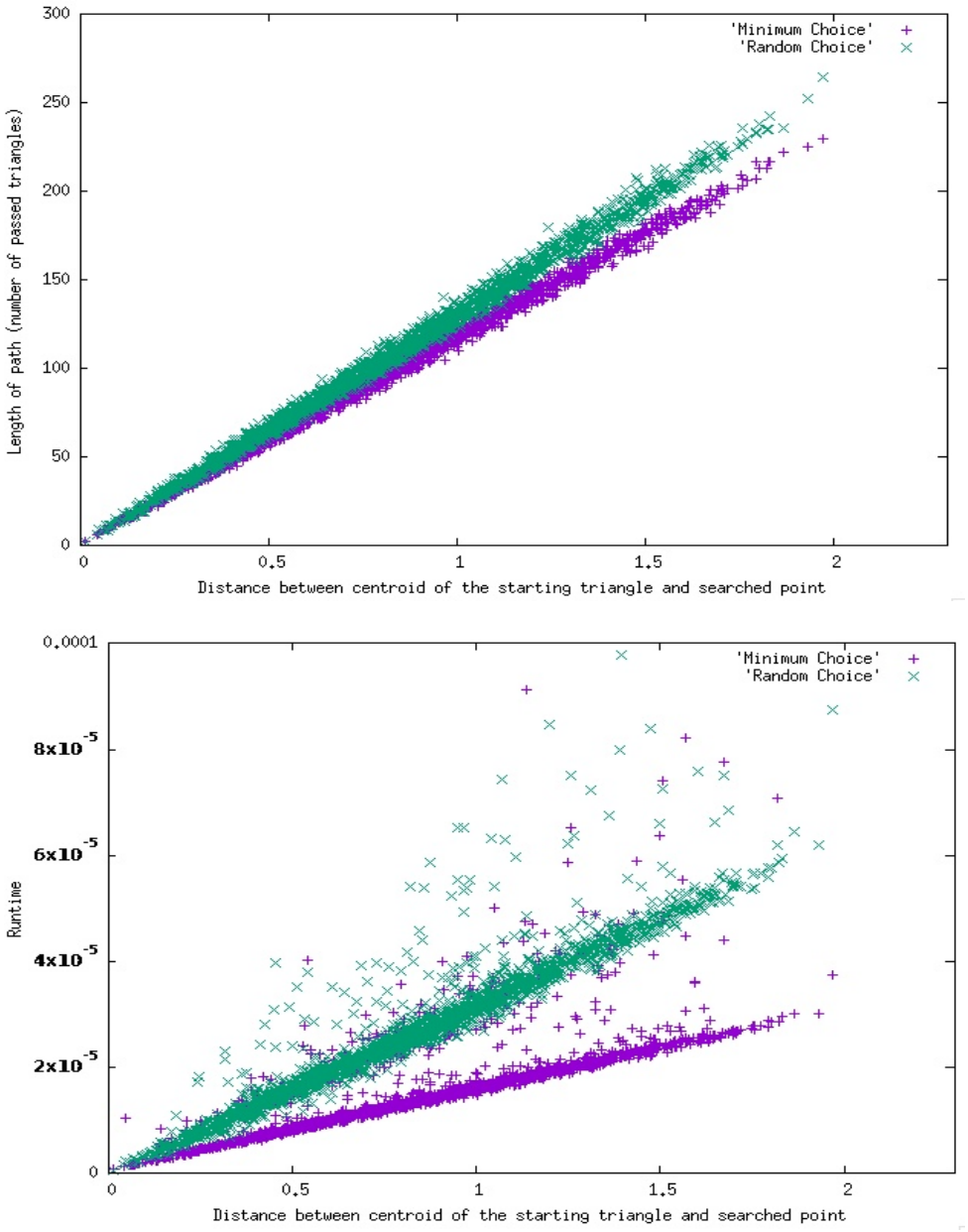


Figure 6: Here, 2000 data points were randomly created.

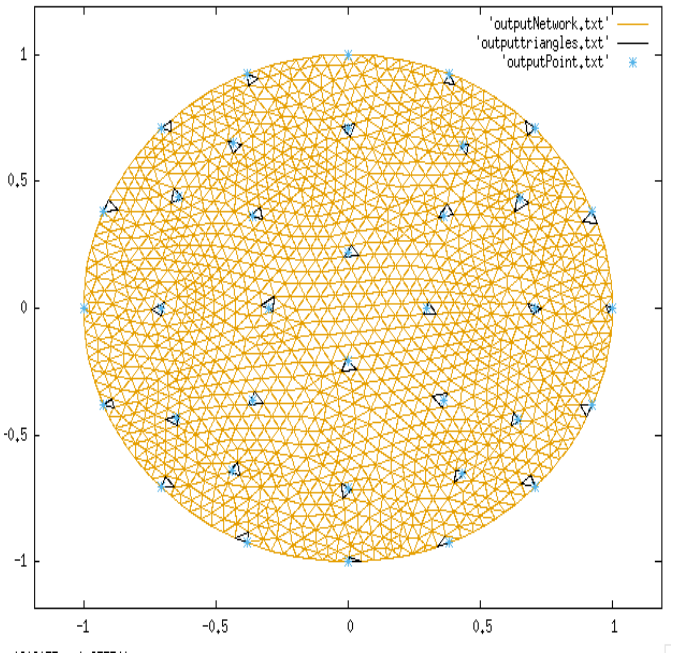
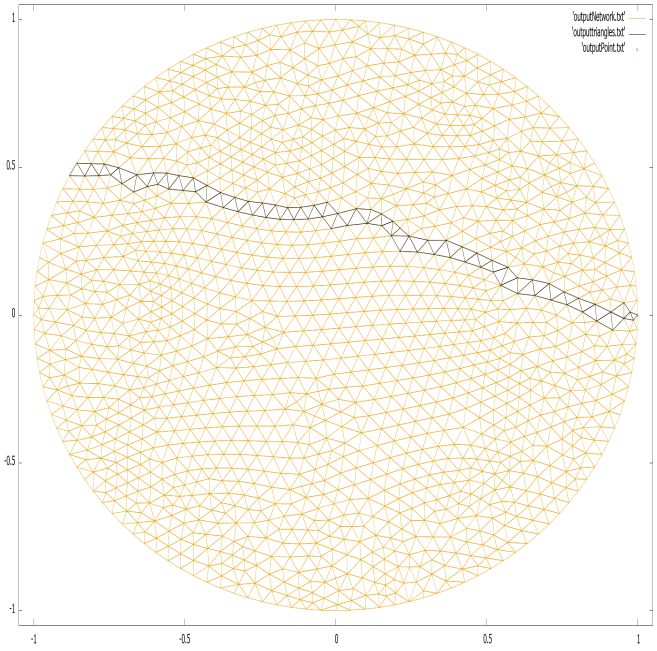
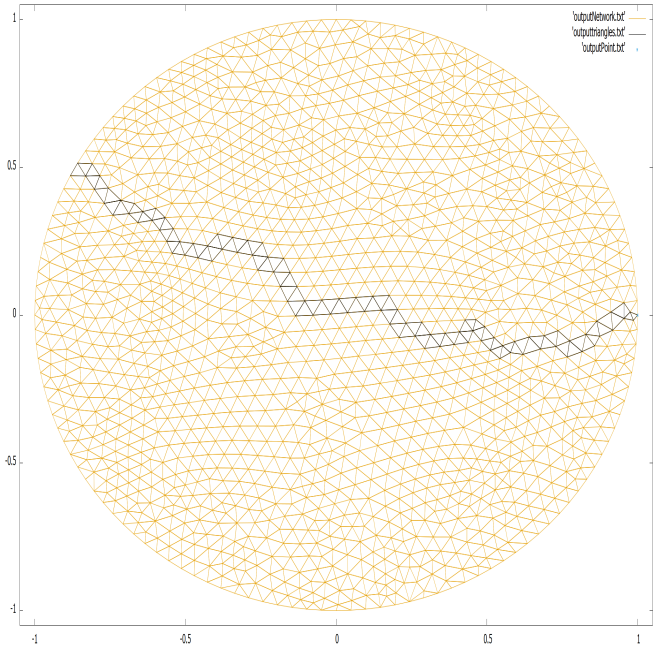


Figure 7: Covering of mesh1.msh by mesh2.msh.





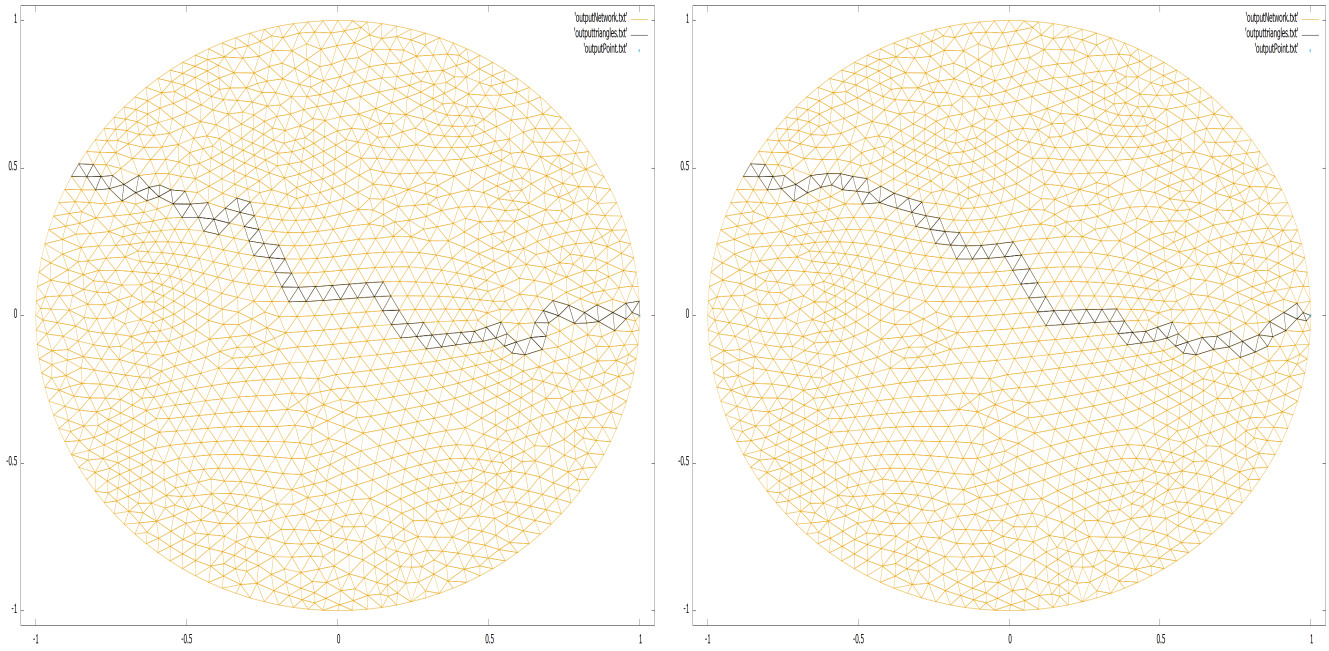


Figure 8: Illustration of path returned by Promenade. The first path was generated by choosing the consecutive triangle using `min_neg` whereas the other three ones are created using `random_neg`.