Université Bordeaux 1
351, cours de la Libération F-33405 Talence cedex

PER PROJECT REPORT

Grid deformation for information visualization

A. Lambert, R. Bourqui, D. Auber

February 7, 2012

Clients:
David Auber
Romain Bourqui

Students:
Barro Lissy Maxime
Manano Camille
Rozar Fabien
Tombi A Mba James
Zenati Omar

Engineering students
enseirb-matmeca 2011/2012

Abstract

The article we worked on deals with how to visualize graphs containing many nodes and edges. With huge amounts of data generally comes visual clutter, in our case due to edge crossing. This solution is based on an edge bundling technique coupled with a grid built from the original graph. Our solution uses the Tutte algorithm in order to quickly obtain a graph without crossing, based on a triangular-face grid, which helps to read relationships between nodes. Moreover, this method will uniformize edge length to ease the visualization. This algorithm is available as a standalone program for Tulip.

CONTENTS

INTRODUCTION

The article [1] talks about how to visualize graphs containing many nodes and edges. Improvements in data acquisition leads to an increase of the size and the complexity of graphs and this huge amount of data generally causes visual clutter, in our case due to edge crossing. For example, it could be interesting to visualize data in fields like biology, social sciences, data mining or computer science, and then emphasize their high-level pattern to help users perceive underlying models.

Nowadays, in the research world, the information is easily represented into graphs to visualize more and more data. However, this huge amount of information prevents the graph from being manually drawn: It explains the need of automatic methods able to generate an appropriate graph with all nodes and edges. Yet this graph may suffer from cluttering, which should be reduced for a better understanding.

Our objective all along this project is to read what has been done before relating to this problem, to provide an objective point of view on those previous works, and propose our contribution. We have implemented a method, then optimized its performances with current technologies (OpenMP, Tulip...) and setted our boundaries.

The first part of this document presents review-related work on reducing edge clutters and enhancing edge bundle visualization, with which the article is connected. The second will deal with the Tutte algorithm and its differents versions. A third part will talk about the implementation issues and show our results. Finally, we draw a conclusion and explain the limits of our work for further improvements.

CHAPTER 1 THE CONTEXT

Some classes of graph, such as trees or acyclic graphs, clearly facilitate user understanding by effective representation. However, most graphs do not belong to these classes, and algorithms giving nice results in terms of time and space complexity but also in terms of aesthetic criteria for any graph do not exist yet. For example, the force-directed method produces pleasant and structurally significant results but does not help user comprehension due to data complexity. The authors of the paper specify two techniques for that reduction: compound visualization and edge bundling. But their interest goes to Edge Bundling, which suggests to route edges into bundles in order to uncover high-level edge patterns and emphasize information. [1, 2] Their contribution was to set this edge bundling by discretizing the plane into a new mesh.

Up to now, several techniques have been used to reduce this clutter, based on compound visualization or edge bundling. In a compound visualization, nodes are gathered into metanodes and inter-cluster edges are merged into metaedges. To retrieve the information, metanodes could be collapsed or expanded. Yet an important constraint is the impossibility for some nodes to move while avoiding edges crossing because node positions provide information: consequently, compound visualization is not suitable. To reduce visual clutter, another clue is to keep vertices while edges are aggregated: the Edge bundling technique routes edges into bundles. This uncovers high level edge patterns and emphasizes relationships. Moreover, some existing representations take into account the the inability of some nodes to change position i.e. some reducing edge clutters (Edge routing, Interactive techniques, Confluent Drawing, Edge clustering) and other enhancing edge bundles visualizations (Smoothing curves, Coloring edges).

The publication we are currently working on is based on a new edge bundling algorithm for efficient graph drawing. By using specific discretization methods such as quad-tree and Voronoï diagrams which are little time-consuming calculations, the authors obtained a new separation of the region where they can reduce the drawing area. As a result, their final discretization algorithm is a good trade off between good precision(*quad tree*) and the computation time ($Vorono$ï)$[ref]$.

In order to create the Edge-bundling effects, the authors use the "shortest path" Dijskstra algorithm. But this does not create a decent number of bundles. Then they add new concepts such as *roads* and *Highways* to increase this number. This means to reduce the weight of an edge (of the grid obtained before) if it is highly used, but only after computing several shortest paths between linked nodes of the original graph.

Several optimizations of the specific shorstest path algorithm such as function calls reductions, multithreading reduces tremendously computation time of graphs.

The authors of the article, after the rendering of their algorithm, think that their grid can be improved because what they have got presents some inconveniences such as the big amount of bends ("zigzag" effect on the grid) and irregular triangles. To solve those problems, they want a heuristic to uniformize sizes of the edges and increase angles between edges that harm graph reading. The proposition in our work is the use of the Tutte algorithm which is only applicable for an internally triangulated planar graph and renders more informative and aesthetic graphs (Less edge crossing, less bends, smaller sizes of edges, maximum angular resolution).

CHAPTER 2 TUTTE

2.1 Tutte's algorithm

The basic graph theory terminology defined in the article [4, 3] will be used. Let $G = (V, E)$ be a planar graph. A mapping Γ of G into the plane is a function $\Gamma : V \cup E \to P(\mathbb{R}^2)$. This function maps a vertex $v \in V$ to a point in $\mathbb{R}^2$ and an edge $e = uv \in E$ to the straight line segment joining $\Gamma(u)$ and $\Gamma(v)$. A mapping is an embedding if distinct vertices are mapped to distinct points and the open segment of each edge does not intersect any other open segment of an edge or a vertex.

A way to build embeddings of any planar, 3-connected graph $G = (V, E)$ have been produced by Tutte in 1963 [6]. Let C be a cycle of vertices. Those vertices are the vertices of a face of G in some (not necessarily straight-line) embedding of G. Let Γ be a mapping of G into the plane.

Theorem 1 *(Tutte's Theorem) Let Ve be a set of the vertices of the cycle C mapped to the vertices of a strictly convex polygon Q, in such a way hat the order of the points is respected. If for each vertex in $V_i = V\ V_e$ is a barycenter with positive coefficients of its adjacent vertices (Tutte assumed all coefficients to be equal to 1, but the proof extends without changes to this case), so Γ is an embedding of G into the plane, with strictly convex interior faces.*

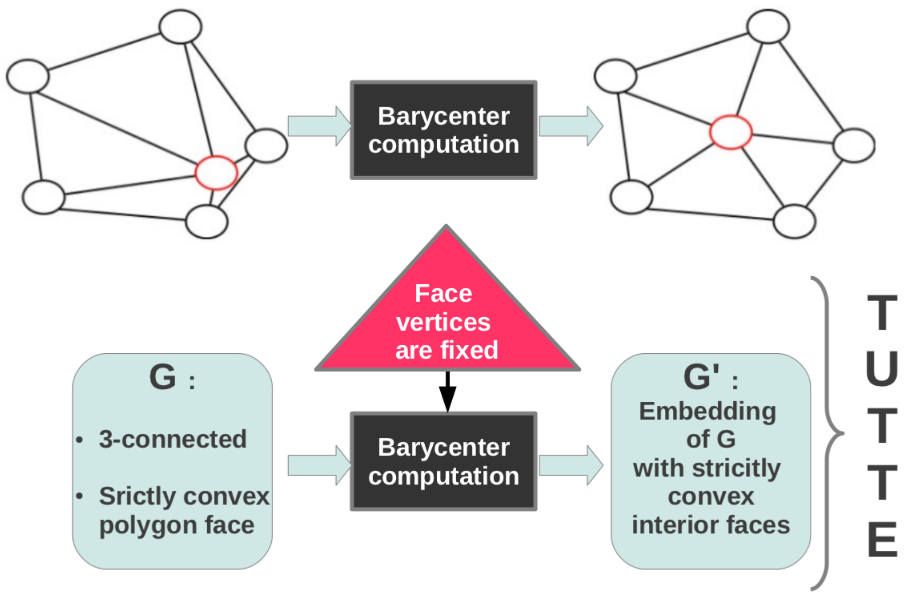

Figure 2.1: Tutte's theorem illustration

2.1.1 Some extreme cases about Tutte's theorem

In our project we do not use 3-connected graph but graph whose interior faces are triangle. It is obvious that considering the kind of graphs we used Tutte's theorem is verified because the hypothesis « All interior faces are tringle » is lighter than « 3-connected » hypothesis. So now we changed the hypothesis about the external polygon and the interior vertices in order to find out if the tutte's is always verified.

Tutte's theorem and concave polygon

In this section, we changed the hypothesis about the graph face. Now we consider a concave polygon face. Below is the illustration of a couterexample.

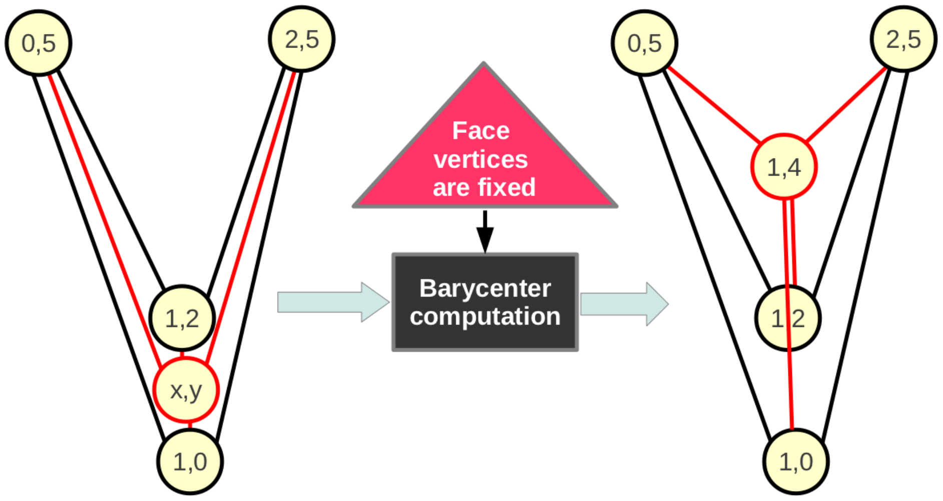

Figure 2.2: Tutte algorithm on concave polygon

In the counterexample illustration above, the coordinate of the vertex shown in red color become

$$(\frac{0+1+1+2}{4}, \frac{5+0+2+5}{4}) = (1,4)$$

One can see that after the barycenter algorithm computation, the graph is no longer embedding. So the Tutte's theorem is not verified considering graphs with concave polygon face.

Tutte's algorithm and convex polygon with some fixed vertices

This section kept the «convex polygon face» hypothesis but some internal vertices are considered fixed, their position never change during the barycenter algorithm computation. Below is the illustration of a couterexample.

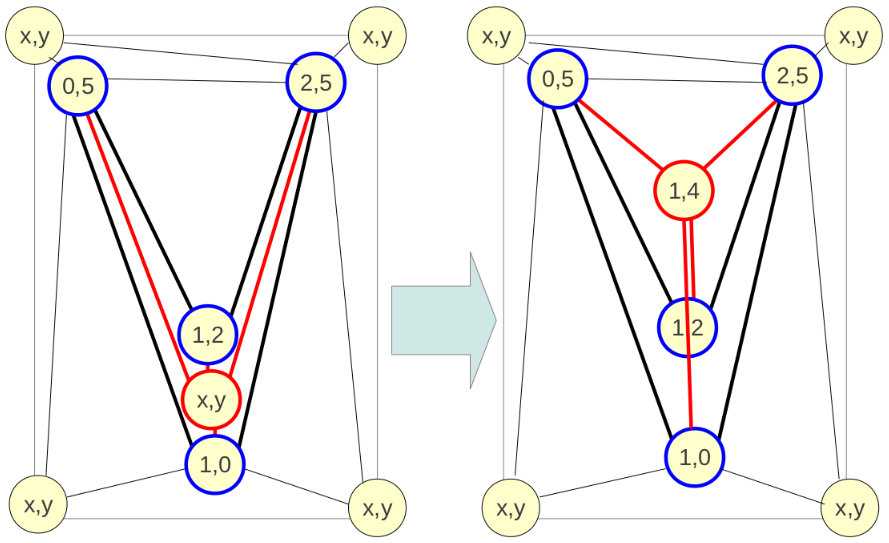

Figure 2.3: Tutte algorithm on convex polygon and fixes vertices

As in the previous counterexample illustration (fig 2.2) above, after the barycenter algorithm computation, the vertex shown in red color position changes. So the graph is no longer embedding which implies that the Tutte's theorem is not verified considering that some internal vertices can be fixed.

2.2 Tutte's Sequential Algorithm

To obtain the graph resulting from the Tutte theorem, an algorithm is needed. In this section, a sequential algorithm is described. This algorithm is an iterative solution. To compute a solution, a set of nodes making up a convex polygon must be decided. Let P be this set of nodes. Let G_k be the graph generated at the step k.

To obtain the graph of the next step, all the nodes of the interior of the convex polygon P will be visited. For each node visited, the barycentric coordinates of its neighbourhood are computed. Then these coordinates are used to update the position of the current node. Once each node has been visited, the computation of the graph G_{k+1} is completed.
In this project, the stop condition used for this algorithm is an epsilon between the relative positions of the node of the graph G_k and G_{k+1}. For all nodes of the graph, if the length of each movement is inferior to a given epsilon, the graph G_{k+1} is the solution.

The figure 2.4 gives an exemple of the mouvement of one node during one step.

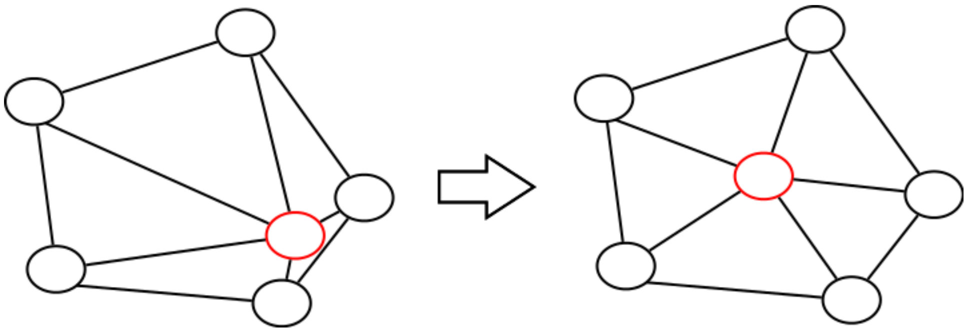

Figure 2.4: barycenter computation of the red node

This gives a synthetic view of this algorithm :

```
G = {V,E}
P = set of nodes constituent a convex polygon
procedure tutte(G, P, epsilon)
  epsilon_current = 0
  for each node of (V \ convex polygon)
    barycenter = barycenter of neighboring nodes
    epsilon_current = max(epsilon_current, distance(node, barycenter))
    node = barycenter
  if (epsilon_current < epsilon)
    exit()
  tutte(G, P, epsilon)
```

From the complexity point of view, during one iteration all the nodes of the set V - *convex polygon* are visited. For each node, all its neighbourhood is visited. Let $mean\_d$ be the average degree of this planar graph. So the compexity of an iteration is $O(n \times mean\_d)$ with $n = |V|$.

2.3 Tutte Parallel Algorithms

In the sequential version, the algorithm presented is an asynchronous Tutte one. In fact, there is a second approach, the synchronous Tutte version. The difference between the two approaches is :

- in asynchronous version: each movement is applied directy after being computed
- in synchronous version: all movement are computed before being applied

In the sequential version, the synchronous approach does not present benefits. It is more convenient to consider neighbors movement.

In the parallel version, the synchronous approach may be interesting in order to reduce critical sections. For the both parallel approach use the OpenMP library.
The complexity these two algorithms is roughly $O(\frac{n}{p})$ with $n = |V|$ and p is the number of processeurs.

2.3.1 Asynchronous parallel version

Distribution of nodes: Graph coloring

In order to implement a parallel asynchronous version of the Tutte algorithm, it is necessary to separate graph nodes into different sets. The objective is to extract an independence between nodes. In fact, each node has to move while the neighbours maintain their positions. Thus, the independence must be between the moving node and its neighbours. This problem is similar to the famous problem of graph coloring.

The objective of the modified Tutte algorithm is to handle graphs of thousands of nodes. To separate such a number of nodes, it is more effective to use a heuristic of the algorithm of graph coloring.
The greedy algorithm is a simple and good solution to separate nodes into sets fast and effectively. [5]

The algorithm used in this project is :

```
G={V,E}
Y = V
color = 0
While Y is not empty
   Z = Y
   While Z is not empty
      Choose a node v from Z
      Colorate v with color
      Y = Y - v
      Z = Z - v - {neighbors of v}
   End while
   color ++
End while
```

This algorithm is known to use at most $d(G) + 1$ colors where $d(G)$ represents the largest value of the degree in the graph G. However, its shortcoming is that it produces sets of different size. This can be inconvenient for task distribution.

Applying Tutte algorithm to sets

Once a set of nodes is obtained, it is possible to apply a parallel Tutte algorithm. The question is how to parallelize it on sets of nodes. The natural idea is to attribute one set per thread : This distribution is far from being optimal. In fact, each thread has to lock the

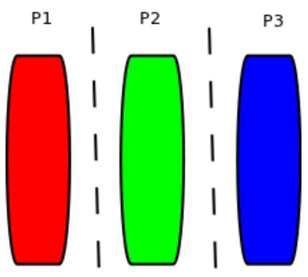

Figure 2.5: One set per thread

neighbours nodes before moving the concerned node, which introduces an important critical section. In addition to being unfair, this distribution is limited by the number of sets produced.

The best distribution for sets obtained by graph coloring is to execute n threads on one set. Each thread moves a number of nodes of the set without any critical section, since each node of the set is not the neighbour of all the other nodes of the same set. Once the thread has moved all its nodes of the set, it must wait for other threads to have completed the same process (implemented by a barrier). Then, the overall process is applied to the next set.

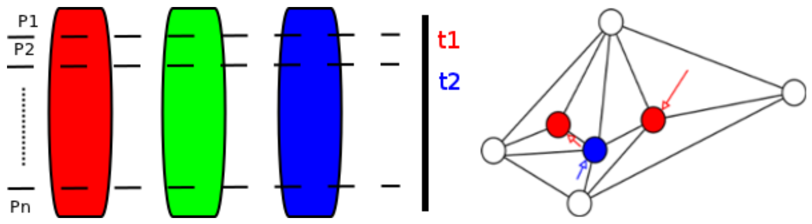

Figure 2.6: n threads per set

2.3.2 Synchronous parallel version

The parallel synchronous version of the Tutte algorithm is easier to implement then the parallel asynchronous. For this version, one does not need to compute a coloring separation.

During an iteration, the new coordinates of all nodes on the graph are saved in a table instead of being directly applied. At the end of an iteration, all upgrades are applied. In this approach, two nodes can be computed independently in parallel.

Consequently, for an iteration, the set of nodes to visit can be separated in different subsets. Each of these subsets is independent and can be computed by different threads in parallel. The separation is static, i.e. the number of element of each subset is equal to *nb_node/nb_thread*.

CHAPTER 3

IMPLEMENTATIONS

3.1 Data Structure

In the implementation of our solution we have defined our own data structure on which we execute the Tutte algorithm. We have implemented some mechanisms to convert a tulip format graph to our own graph structure and also to get information from our structure to insert them into a Tulip graph. In other words, our structure is a temporary structure for storing information about nodes in order to execute the Tutte algorithm.

3.1.1 Issues

As the Tulip data structure contains a lot of information, it is expensive to manipulate them. Furthermore, we do not need all the information from a given Tulip graph. For instance, for a given node, we just want to know if it is fixed. For a fixed node, position never changes during the Tutte algorithm. In addition to that, as we are looking for performance, we need a light structure matching the principe of Tutte algorithm. The following points are the main reasons which lead us to set up a new data structure.

1. The fact that a given node is fixed or not is indicated firstly by a mobility property. However, there is another property indicating nodes which are part of graph contouring, and these nodes need to be fixed too. Therefore, to deal with the fact that a given node is fixed or not, we need to manipulate two properties that cost a lot.

2. In Tulip data structure there is a hierarchy of graphs. However, we only need the parent of the graph. We do not need the sub-graph relation between graphs.

3.1.2 Implementations

We tested three implementations in order to find out the right one. Because we care of memory and speed, we merely store only the information needed to run the algorithm in our structure.

First implementation

In this implementation, our structure is constructed so that a given node contains its neighbourhood. So one can easily access the neighbourhood of a given node because it is very crucial in a Tutte algorithm implementation. To do this, we define a class that contains the various data needed on a given node (the attributes) and all the operations we need to run on a node (the methods).

```
class MyNode {
 private:
  node n;
  bool mobile;
  Coord coord;
  vector<MyNode *> voisin;

 public:
  MyNode();
  MyNode(const node n, const Coord coord);
  MyNode(const node n, const bool mob, const Coord coord);
  ~MyNode();

  const node getNode() const;
  bool getMobile() const;
  void setMobile(const bool b);
  const Coord getCoord() const;
  void setCoord(const Coord &);
  vector<MyNode *> * getVoisin();
  vector<MyNode *> getVoisin() const;
};
```

Vertex attributes needed

n : `node` type of Tulip library; contains the `ID` of the node.

mobile : `boolean` type; is used to know a given node is considered fixed.

coord : `Coord` type of Tulip library; is used to store the node coordinates.

voisin : `vector` type of `C++` library; contains the neighbourhood.

Operations on a vertex

We used two types of operations or methods: setter and getter. A setter is a method used to set the value of an attribut and a getter is used to get the value of an attribut. For a given attribut `attribut` , the corresponding setter and getter are respectively `setAttribut(args)` and `getAttribut()`. Below are the lists of the setters and getters of nodes in our structure:

Setters : `getMobile(), getCoord(), getVoisin()`

Getters : `setMobile(const bool b), setCoord(const Coord &), getVoisin().`

Second implementation

In the second implementation, the built data structure aims to decrease the number of pointer translation of the system. Also, the method to fill our data structure tries to put the neighbourghood coordinates of a node near to its own.

A new basic class of MyNode_ver2 have been implemented. The vector `vector<MyNode *> voisin` is substituted by an integer `index_neighbourhood`.

Here is the statement of this class:

```
class MyNode_ver2 {
  public:
  node n;
  bool mobile;
  int index_neighbourhood;
  int degree;
};
```

To store all the nodes, the neighboughoods and the coordinates, three tables are needed.

```
vector<MyNode_ver2> MyNodes_2;
vector<int> Neighbourhoods;
vector<Vec2f> * coords;
```

MyNodes_2 : contains all the nodes.

Neighbourhoods : contains the index of all the neighbourhoods.

coords : is used to store the nodes coordinates.

The attribute `index_neighbourhood` of the class `MyNode_ver2` gives the index of the neighbourhood of the node in the vector `Neighbourhoods`.

Third implementation

In this third implementation, we do not use a class to store the various data about node to run Tutte algorithm. As we are looking for a lighter data structure in order to ameliorate memory access, we use a `struct` to group data needed about a given node under one name (`Data`).

```
struct Data {
  node n;
  Coord coord;
  bool mobile;
} Data;
```

In addition of the structure above, we use two tables: a `data store table` table to store data about nodes and `neighbourhoods table` to link nodes with their neighbourhoods. The picture below illustrate the principle.

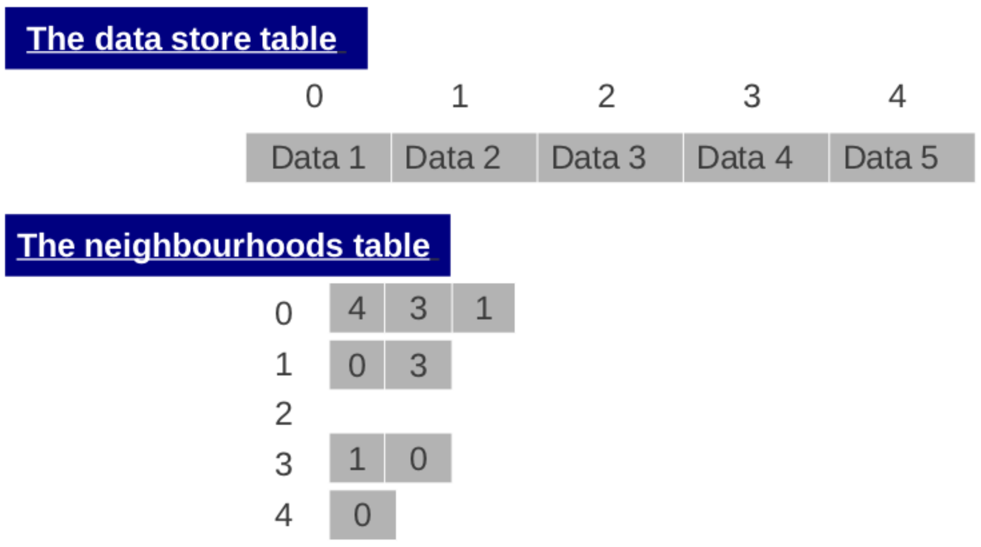

Figure 3.1: Third implementation graph representation

One can read in the `neighbourhoods table` that neighbours of the node 0 are nodes `4, 3, 1` and node 2 does not have neighbours. One can access all information about node 0 located at the index 0 of the `data store table`.

3.2 Results

3.2.1 Benchmark

The authors provided us with three graphs in order to test our different implementations of the Tutte method.

These graphs have the following characteristics :

Graph	number of vertices	number of edges
aiir_traffic	14693	63403
imdb	9488	33942
migration	14318	49460

The histogram (figure 3.2) shows the different times of execution that we obtain through our different implementations:

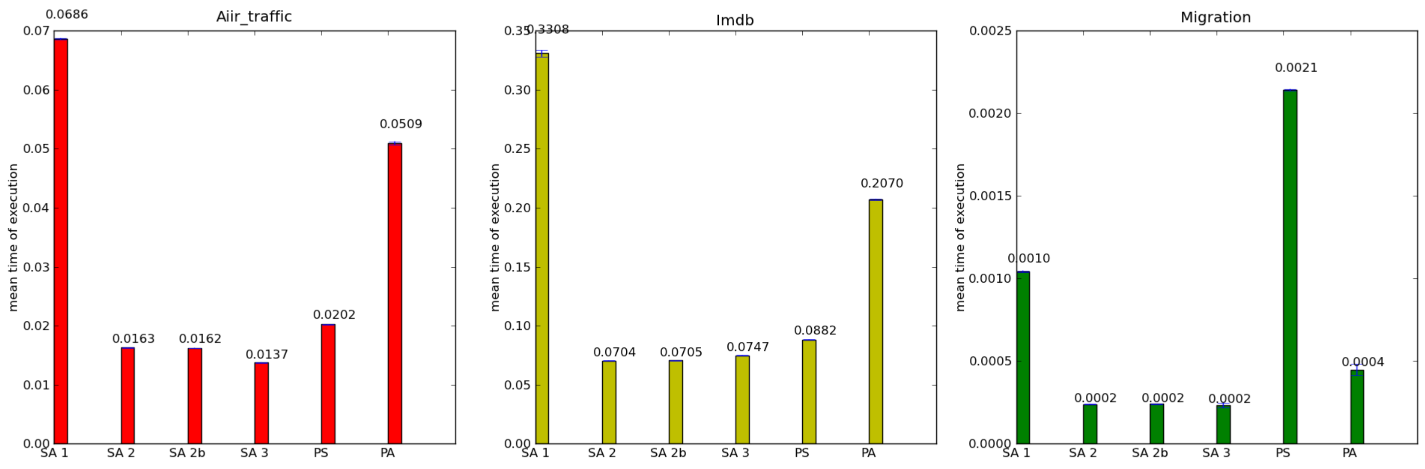

Figure 3.2: All the mean times of execution

Definition of each tag above a bar

SA 1 : Tutte sequential asynchronous version on the first data structure;

SA 2 : Tutte sequential asynchronous version on the second data structure;

SA 2b : Tutte sequential asynchronous version on the second data structure with usage of Vec2f;

SA 3 : Tutte sequential asynchronous version on the third data structure;

PS : Tutte parallel synchronous version on the second data structure;

PA : Tutte parallel asynchronous version on the first data structure;

The mean times of computation showed on the figure 3.2 have been obtained with over 1000 executions of each algorithm. The configuration of the computer having done the executions is :

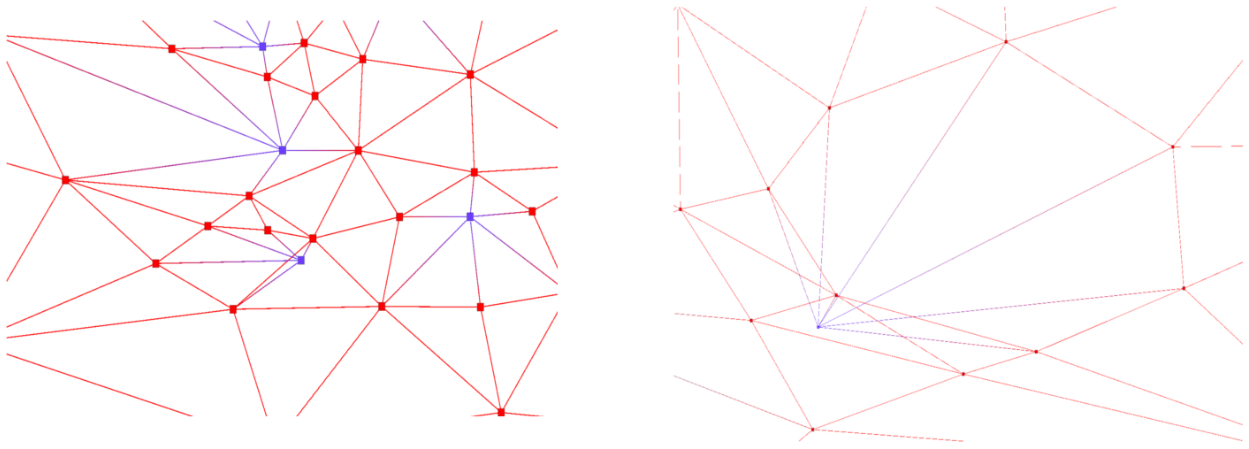

Figure 3.3: Overview of edge crossing on Aiir Traffic graph

Processor: Intel Core i5-2410M

RAM: DDRIII 6 GB

One can notice that the best implementation is the sequential asynchronous version on the second data structure. Also, the parallel implementations are not the faster, certainly because the number of data to treat is not great enough.

3.2.2 Not planar graph

The results we get here are not as good as we expected. Although the computation time is quite improved, the produced graphs are not planar. The reason is that the provided graph has some fixed nodes inside the grid. Consequently, after the call of our implementation of the Tutte Algorithm, some edges crossing (due to those fixed nodes) appear and remove the planar property of the graph. As an explanation, we can say that moving nodes tend to be oriented to the side which has the highest level of fixed nodes.(see the figure 3.4). With such a graph, a mobile node which is opposite to the side with numerous edges will move to this side and create an edge-crossing (see the figure 3.5).

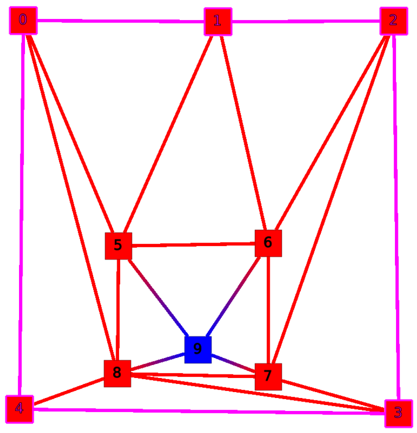

Figure 3.4: The initial graph. The blue node is fixed

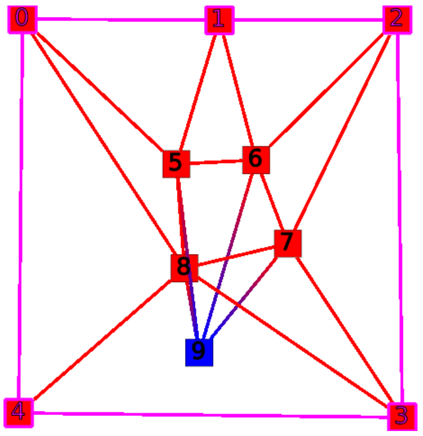

Figure 3.5: The set not correctly modified

You can notice that if all the nodes in the interior of boundaries are mobile, then the graph produced stay planar (see figure 3.6 and 3.7).

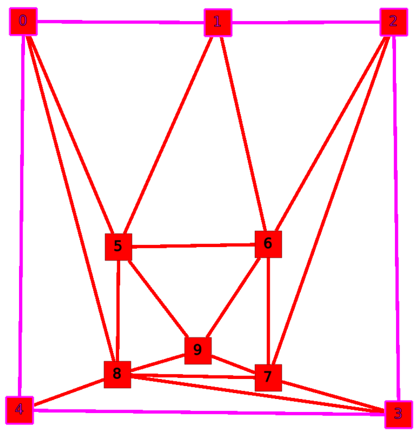

Figure 3.6: The initial graph with all interior nodes mobile

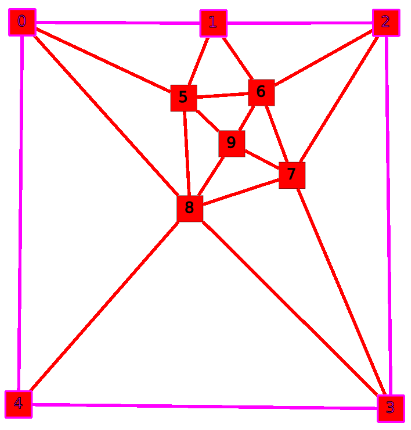

Figure 3.7: The graph correctly modified

CONCLUSION

After understanding problems due to the grid building, we proposed the Tutte algorithm to increase data comprehension and ease graph reading. This permits us to obtain uniformized size triangles and increase angles between edges while avoiding edge crossing. Finally, our method allowed an improvement of the clutter reduction and help unterstanding. Furthermore, thanks to our optimizations, the performance is comparable to existing methods.

As explained below, the Tutte algorithm only works in some specifics cases. We showed it may fail for a concave polygon or when some nodes are fixed. However, the grid used is particular: it is created by applying a quad-tree algorithm, then a Voronoi decomposition and has the following constraint: two fixed nodes can not be linked by an edge. Unfortunately, despite these grid characteristics, we also discovered that the apply of the Tutte algorithm on this graph leads to edge crossing, which breaks the planarity of the final graph. This is because the grid used may contain some fixed nodes created during the grid building.

In future work, this problem could be solved by the uniformization of the triangle size, i.e. automatically join small triangles or divide big triangles into smaller ones. It is nevertheless possible to create a post-treatment function to fix edge crossing. Some others issues concerning our program remain unsolved and could be taken into account in the future. Rather than work with input graph nodes, it would be interesting to dynamically add nodes (and their edges to keep an internally triangulated planar graph) and to launch again the algorithm in order to refine the given results.

BIBLIOGRAPHY

[1] A. Lambert, R. Bourqui, and D. Auber. Winding roads: Routing edges into bundles. *In 12th Eurographics/IEEE-VGTC Symposium on Visualization (Computer Graphics Forum; Proceedings of EuroVis 2009).* To appear., 2010.

[2] A. Lambert, R. Bourqui, and D. Auber. 3D edge bundling for geographical data visualization. *In IV '10: Proceedings of the 14 International Conference on Information Visualisation (IV'09)*, Washington, DC, USA, 2010. IEEE Computer Society.

[3] B. Bollob's. Modern graph theory, *volume 184 of Graduate Texts in Mathematics*. Springer-Verlag, 1998.

[4] E. Colin de Verdière, M. Pocchiola, and G. Vegter. Tutte's Barycenter Method applied to Isotopies. *Computational Geometry: Theory and Applications, 26*, 81–97, 2003.

[5] Gormen, T.H. and Leiserson, C.E. and Rivest, R.L. and Stein, C. Introduction to algorithms. *In MIT press Cambridge, MA*, 16:"Greedy Algorithms", 1990.

[6] William T. Tutte. How to draw a graph. *Proceedings of the London Mathematical Society*, 13:743–768, 1963.