Tulip – A Huge Graphs Visualization Framework*

David Auber

LaBRI-Université Bordeaux 1, 351 Cours de la Libération, 33405 Talence, France

1 Introduction

The research by the information visualization community ("InfoViz") show clearly that using a visual representation of data-sets enables faster analysis by the end users. Several scientific reasons explain these results. First of all, the visual perception system is the most powerful of all the human perception systems. In the human brain, 70% of the receptors and 40% of the cortex are used for the vision process [27,34]. Furthermore, human beings are better at "recognition" tasks than at "memorization" tasks [10]. This implies that textual representations are less efficient than visual metaphors when one wants to analyse huge data-sets. This comes from the fact that reading is both a memorization task and a recognition task.

The research by Bertin [5], Fairchild [12] and Ware [34] show that two types of information exist : entities and relations. Thus, all kinds of data-sets can be represented by a graph with attributes. In this graph, the nodes are the entities and the edges are the relations. The graph visualization framework that we present in this chapter has been set-up in order to devise, experiment and use, for the information-visualization purpose, new tools based on the results coming from the graph theory. For instance, one of our results [4] consists in using the statistical properties of random trees having a fixed maximum-degree or/and a fixed maximum segment-length in order to detect automatically the irregularities in the visualized data-set.

Even if the Tulip framework enables the visualization, the drawing and the edition of small graphs, all the parts of the framework have been built in order to be able to visualize graphs having up to 1,000,000 elements. When one wants to visualize such structures, interaction is necessary. Ideally, the human perception system requires an interactive system to respond to an operation in less than fifty milliseconds [34]. Such time constraints imply that a special emphasis should be layed on the complexity of the algorithms. Furthermore, when the size of graphs becomes huge, both due to the limits and the architecture of existing memories, reducing the memory consumption is one of the most important requirements.

Moreover, if we follow the information retrieval method introduced by Shneiderman [26] :

* We gratefully acknowledge Marie Cornu, Michael Jünger and Petra Mutzel for their constructive remarks on earlier drafts.

"Overview first, zoom and filters, then details on demand",

a visualization system must draw and display huge graphs, enable to navigate through geometric operations as well as extract subgraphs of the data and allow to change the representation of the results obtained by filtering [20]. Thus, a graph visualization system must allow three things : graph drawing, graph clustering and interaction. The Tulip framework is especially designed to support such a visualization method on huge graphs.

This chapter presents some of the direct applications of the Tulip framework in research and bioinformatics. Then, it introduces different kinds of algorithms implemented in Tulip such as : graph drawing, graph parameters and graph clustering. Subsequently, we will describe the general functioning of the Tulip data structure that enables the management of a graph hierarchy and the manipulation of algorithms; we will also give a short description of the Tulip [1] software functioning. Finally, after giving several examples of graph drawing results obtained on real graphs, which come from the filesystems, the web site and the program analysis, we will conclude with general information about the terms of use of the framework.

2 Applications

2.1 Research

The first application of the Tulip graph visualization framework is to provide an environment in which tools can be experimented for the purpose of the information visualization research. Thus, our team and others are using it in order to devise new kinds of human computer interfaces and new tools. Huge data-sets can consequently be manipulated. Using graphs for information visualization is a complex task in which graph drawing algorithms take an important place. The most important pieces of research currently supported by the Tulip framework are : using graph parameters in order to show up information in data [3,4], using graphs in order to make a visual analysis of semantic networks [6], and studying general new concepts of data exploration. These different pieces of research take place in the university of Bath (UK) , the university of Chicoutimi (Canada), Virginia tech (USA), the university of Montpelier (France) and the university of Bordeaux (France).

2.2 Biochemistry

One of the direct applications of the Tulip framework can be observed in a European project that consists in giving access to information about some protein-protein interactions to biologists. The protein-protein interaction(PPI) data are developing into increasingly important resources for the analysis of metabolic pathways, signal transduction chains and other cellular mechanisms. This allows, for instance, to find specific drug treatments to inhibit or

activate a precise and selected metabolic pathway. Therefore, it decreases side effects. One reason for this current increase in the utilization of technologies which yield the PPI data is the fact that these technologies, in particular the yeast two-hybrid technology, are relatively well-studied for the development of high throughput experiments conducted by different groups. However, the comparisons of results obtained by different groups is currently very labor-intensive. Thus, one of the applications of the Tulip framework is to use the graph visualization method in order to simplify the comparisons of results through an intuitive human computer interface that uses graph visualization methods and interaction with the data.

2.3 Others

The architecture of the software has been split into different parts that can be reused in order to build new graph visualization applications easily. Thus, the use of this framework into direct applications is growing quickly. For instance, it is used in order to display the results of an automatic analysis of videos in the MEPGS format in which the generated data are graphs and trees. Another applications is for the visualization and the comparisons of sequences and secondary structures of DNA, in which graph drawing algorithms are used for an automatic displaying of the data and the combinatoric properties of the trees are used to simplify the comparisons.

3 Algorithms

Since the Tulip software is dedicated to graphs visualization, several kinds of algorithms are available. The most important ones are : graph drawing algorithms, clustering algorithms, metrics algorithms and visual attribute mapping algorithms. Afterwards, we will present some of the algorithms available through the Tulip framework.

3.1 Graph drawing

As presented in the chapter "[TF]", we know that a large variety of graph drawing algorithms dedicated to graphs, directed acyclic graphs and trees already exists. Here we will focus on some algorithms that can apply for a huge "real" graphs visualization. When we consider huge graphs, time complexity is very important and as a consequence, a special emphasis should be layed on the minimization of the quantity of memory used in all the algorithms. This necessity comes from the fact that, in most of cases, even if an algorithm is linear, the access to the memory on the hard disk drive (swap) makes the algorithm unusable. One of the reasons is that a graph structure is by definition not linear, thus, the memory management algorithms (like cache, or prefetch) are inefficient most of the time.

Afterwards, we will introduce a different Reingold and Tilford [24] algorithm which enables to reduce the memory consumption and which permits to include nodes of arbitrary individual size and edges of arbitrary individual length. Then we will introduce the Cone Tree algorithm originally designed by Robertson et al [25] for the filesystem visualization. This algorithm is of general interest for the purpose of information visualization. We will detail one of its improvement by Carriere and Kazman [9] and then we will present an heuristic that produces better drawing if the tree has a non-constant node degree. To finish, we will present a graph drawing algorithm, which produces Sugiyama-style layout, that is especially designed to handle huge graphs. This algorithm draws graphs with at the most two bends per edges, its complexity is in $O([V|\cdot|E|)$ time and $O([V|+|E|)$ space. Other algorithms such as the so-called GEM algorithm of Frick et al [14], the well-known Tutte algorithm [31] and the Tree map algorithm [7] are also available in Tulip.

Reingold and Tilford : As described in "[TF4.1]", the original algorithm of Reingold and Tilford [24] is a recursive algorithm that consists in drawing separately each subtree of a node and then move each subtree until the distance between the subtrees is minimal. The algorithm described by Reingold and Tilford need to store data inside nodes in order to make all the operations.

In order to obtain a usable drawing in terms of information visualization, one must produce a layout in which the node, of individual size, and the edge, of individual length, are taken into account. Our tree drawing algorithm manages the node height by fixing the height of a layer, L, to be the maximum height of a node in L. The node width is taken into account by managing the left and right contours for each node. This algorithm is usual and can be done with slight modifications of the original Reingold and Tilford algorithm. The method used in order to manage the edge length consists in an augmentation step which adds nodes and edges to the tree. Figure 1 summarizes the augmentation method. In the following, we note T as the tree obtained after augmentation.

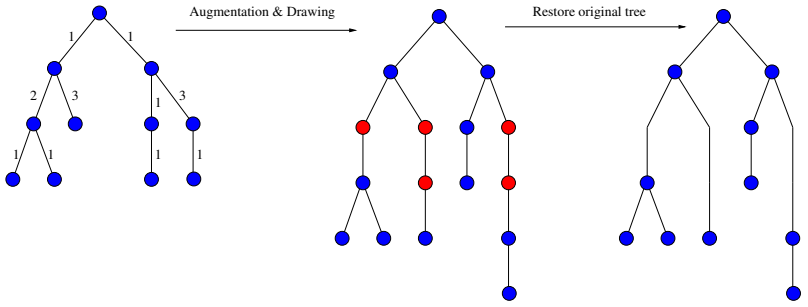

Fig. 1. Edge length management.

Let $h : E \to \mathbb{N}^+$ be a function that represents edge length. If one uses the standard Reingold and Tilford algorithm the time complexity of the algorithm is $O(\sum_{e \in E} h(e))$ and the space complexity is $O(\sum_{e \in E} h(e))$ because one must build a tree of $1 + \sum_{e \in E} h(e)$ nodes and $\sum_{e \in E} h(e)$ edges.

When one wants to draw huge trees, the memory requirement of the algorithm presented above makes it unusable if the edge length is important. However, if we look carefully at the Reingold and Tilford algorithm, we can notice that in order to compute the layout we only need to store two contours at the same time and not two values inside each node like in the algorithm proposed by Reingold and Tilford. Figure 2 illustrates this property. Another important property of the Reingold and Tilford algorithm is that each node (different from the root) of degree equal to two is lined up with its neighbors. Therefore, it is not necessary to store the coordinates of each virtual node added during the augmentation step. Furthermore, the contours induced by an edge having a length greater than one can be stored efficiently. This enables to preserve the memory usage when having a lot of different edges lengths. If we take into account these two considerations, it is possible to add dynamically, and efficiently, the virtual nodes during the contour building of each subtree and then, when the contour is no more used, to free all the unused virtual nodes. The results consist in an algorithm with a time complexity in $O(\sum_{e \in E} h(e))$ and with a memory complexity in $O(|V|)$.

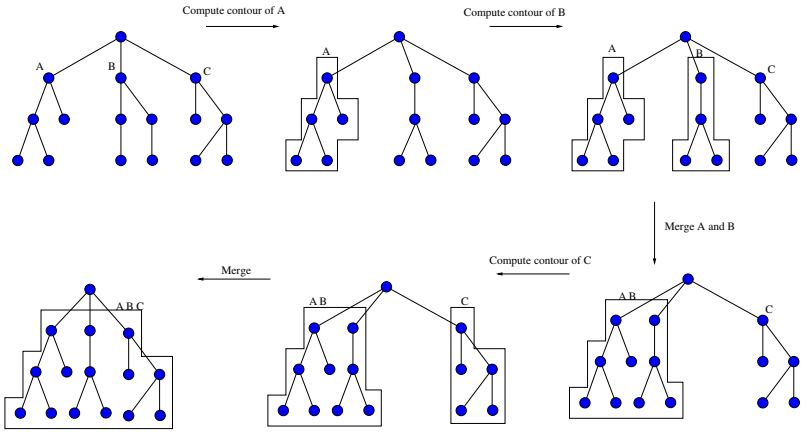

Fig. 2. Tree contours.

The hierarchical tree drawing algorithm proposed in Tulip is the one presented above. For instance, the storing, the drawing and the displaying of a complete binary tree of depth 20 (2^{20} nodes and $2^{20} - 1$ edges) with all edges

having a length of 200.000 and a random size of nodes, requires less than 400 MB of memory and takes 15 seconds [1].

Cone Tree : The "Cone Tree" algorithm was proposed by Robertson et al [25]. It enables to draw general trees in three dimensions. One of its interests for the information visualization purpose is that it allows to represent, in a comprehensible manner, more pieces of information than the two dimensional algorithms.

Carriere and Kazman [9] have shown that, in practice, if one wants to obtain a layout of quality with the Robertson et al [25] algorithm, the degree of nodes must be constant. It comes from the fact that Robertson et al algorithm assigns the same amount of space to each subtree without taking their sizes into account. Thus, when two subtrees have a different size (a different number of nodes), they have the same size in the final drawing. The solution proposed by Carriere and Kazman consists in using a divide and conquer algorithm. Similar in concept to the one of Reingold and Tilford, the algorithm draws each subtree inside an enclosing circle and then places each enclosing circle at a minimum distance on a circle. The third coordinate of nodes is obtained by using their depths in the tree. The principle of the recursive algorithm is the following :

Let $T_D(x)$ be the drawing of the subtree induced by node x, we note $R_{\text{hull}}(x)$ the radius of an enclosing circle of $T_D(x)$. Let s be a node, to simplify, we note s_i the i^{th} elements of $adj^+(s)$. In order to compute $T_D(s)$, the algorithm first computes C_s^p that is the perimeter of the circle C_s on which each $T_D(s_i)$ will be placed :

$$C_s^p = 2 \cdot \sum_{s_i \in\ adj^+(s)} R_{\text{hull}}(s_i)$$

Then, for each i in $[1..deg^+(s)]$, it computes the angular sector $\theta(s_i)$, which corresponds to the space reserved for $T_D(s_i)$, by using the following formula :

$$\theta(s_i) = \frac{4 \cdot \pi \cdot R_{\text{hull}}(s_i)}{C_s^p}$$

Subsequently, it computes angular positions, which will allow to place each subtree drawing on C_s, by using the following recurrence :

$$\theta_{s_1}^{pos} = 0$$

$$\theta_{s_i}^{pos} = \theta_{s_{i-1}}^{pos} + \frac{\theta(s_{i-1}) + \theta(s_i)}{2}$$

Finally, the algorithm computes the enclosing circle of the new drawing by using the following formula :

$$R_{\text{hull}}(s) = max(\{R_{\text{hull}}(s_i) | s_i \in adj^+(s)\}) + \frac{C_s^p}{2 \cdot \pi}$$

[1] On a x86 computer running at 1.5Ghz.

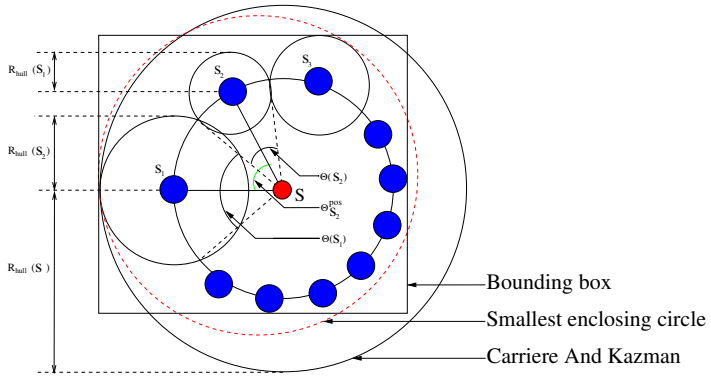

Fig. 3. Carriere and Kazman algorithm.

Figure 3 summarizes the algorithm. On this figure one can see that the enclosing circle used by the Carriere and Kazman algorithm is far from being optimal (id. the smallest enclosing circle). It is straightforward that minimizing the size of the enclosing circle has the effect of reducing the drawing size and of providing a better angular resolution in the final layout. Therefore, the algorithm set-up in Tulip is different than the Carriere and Kazman one. The most important difference is that the center of the enclosing circle can be different from the nodes positions. This change is necessary if one wants to find a better enclosing circle. Finding the optimal enclosing circle is a particular case of a more general problem, called the "Smallest enclosing ball problem". This problem can be formulated as a convex optimization problem. For more information on this subject you can refer to [16,22,35].

In our algorithm, we have set-up a fast approximation. It uses the fact that one can compute directly the optimal enclosing circle of two circles. Thus, it is possible to compute an enclosing circle incrementally by merging each time two circles. It enables us to obtain an enclosing circle of $T_D(s)$ in $O(deg^+(s))$ time. The fact that all circles are placed on a circle at a defined angular position is taken into account to choose in which order the merging operations are done. In the final version of our heuristic we also used as a starting circle the internal circle of the bounding box of all subtrees. The Carriere and Kazman heuristic is optimal if the enclosing-circle size of each subtree is the same. That is why, we compute the enclosing circle obtained by both methods and then we choose the one that has the minimum size.

Experimentations have shown that the results obtained on trees having a non-constant degree are better than the ones obtained by using the method suggested by Carriere and Kazman. For instance, we have measured that when the degrees are chosen randomly the average area of an enclosing circle is 40% smaller than the area obtained by the Carriere and Kazman algorithm. Figures 4 and 5 show the differences of results obtained on a 120.000 nodes tree representing an entire Linux filesystem. On these two figures one can

notice that our method improves significantly the angular resolution and the drawing size.

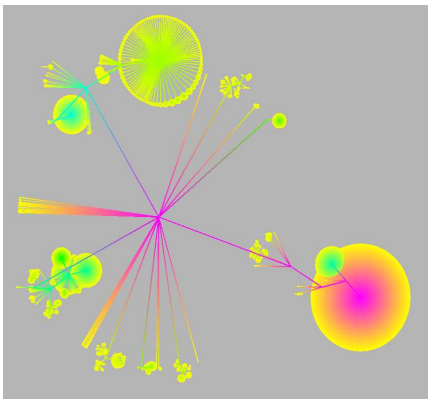

Fig. 4. Carriere and Kazman filesystem drawing

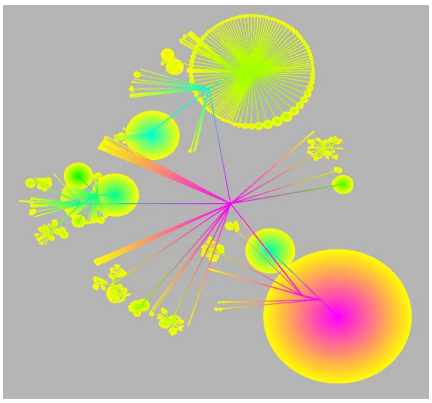

Fig. 5. Improvement of the Carriere and Kazman algorithm

Like with the Reingold and Tilford algorithm, the linear running time is archieved by delaying all the placements of the subtrees in a second phase. For example, the implemented version in Tulip, enables to draw and to display trees such as the ones of figures 4 and 5 in less than one second [2].

Hierarchical Drawing : The advantages of the Sugiyama approach [29] are wide and its usefulness for information visualization is well-known. However, the complexity of the existing algorithms in terms of memory consumption makes them unusable for the drawing of huge graphs. The problem comes from the augmentation step that can add in the worst case $O(|V| \cdot |E|)$ nodes. Even, when using fast Sugiyama algorithms, such as the one proposed by Buchheim et al [8], the problem is still the memory usage. For instance, a drawing of a complete DAG having 200 nodes and 19.900 edges requires 900MB. Figure 6 shows a typical graph in which the augmentation step requires a quadratic number of insertions. In order to allow a huge graph hierarchical drawing that prevents the edges superposition, that reduces the crossings and uses at most two bends per edges, we have set-up an algorithm that uses only $O(|V| + |E|)$ [2] memory in the worst case. Subsequently, we will consider that we manipulate a directed acyclic graph with one source. Using the well-known preprocessing step of the Sugiyama algorithm, one can always transform a general graph in such a graph (cf. "[TF4.2]").

[2] On a x86 computer running at 1.5Ghz.

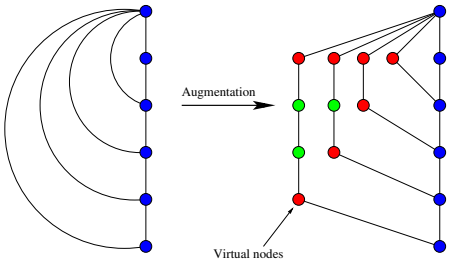

Fig. 6. Quadratic augmentation.

The layer assignment step of our algorithm is done by using the longest path algorithm (cf. "[TF4.2]"). This algorithm is a standard topological walk in a DAG. Let k be the number of layers and $L(x)$ be the layer number of the node x. This algorithm ensures that a path P in the graph between the layer one and the layer k, in which each edge (u,v) verify $L(u) = L(v) - 1$, always exists. Thus, the depth of a spanning tree after the Sugiyama augmentation step is equal to the depth of a spanning tree of the original DAG.

The augmentation step is done by replacing the edges of the original graph and by building a function $E_h : E \to \mathbb{N}^+$. Let $\lambda(s,d) =\mid L(s) - L(d) \mid$. The transformation method of an edge $e(s,d)$ is the following :

$$e(s,d) \to \begin{cases} if\ \lambda = 2, & then\ we\ add\ 1\ node\ u_1\ and\ 2\ edges\ e_1 = (s,u_1), \\ & e_2 = (u_1,d)\ and\ we\ set\ E_h(e_1) = E_h(e_2) = 1. \\ if\ \lambda > 2, & then\ we\ add\ 2\ nodes\ u_1,u_2\ and\ 3\ edges \\ & e_1 = (s,u_1), e_2 = (u_1,u_2), e_3 = (u_2,d)\ and\ we\ set \\ & E_h(e_1) = E_h(e_3) = 1\ and\ E_h(e_2) = \lambda. \end{cases}$$

Furthermore, we assign to the nodes added the following layer number : $L(u_1) = L(s)+1$ and $L(u_2) = L(v)-1$. This method avoids doing a quadratic augmentation and ensures the insertion of at the most $O(2 \cdot |E|)$ virtual nodes in the graph (id. we do not insert green nodes in figure 6). Then, in order to reduce the number of crossings we use the barycenter heuristic (cf. "[TF4.1]"). By default, the number of up and down sweeps is fixed to 4.

In order to make the coordinate assignment step, we first extract a spanning tree. For each node s, the building of a spanning tree consists in removing all the edges in $star^-(s)$ instead of one. Let v be a node, in order to obtain a drawing on which the nodes are placed in the center of their ancestors, this edge is chosen so that the barycenter value (obtained during the crossing minimization step) of its source s is the median of the barycenter values of the elements in $adj^-(v)$. Figure 7 shows the effect of this choice on a simple layout.

The final layout is obtained by drawing the spanning tree with the variant of the Reingold and Tilford algorithm introduced above, in which we use E_h

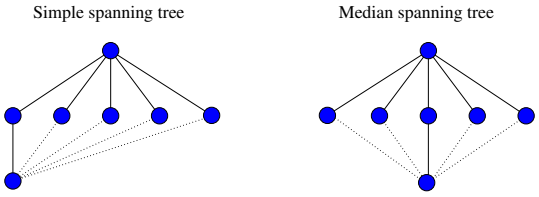

Fig. 7. Spanning tree building.

for the edge length and user-defined sizes of nodes. The result of the algorithm is an upward forward drawing with at the most two bends per edge that need $O(|V| + |E|)$ space and $O(|V| \cdot |E|)$ time. The proof of the space complexity is done by using the property of the longest path algorithm [2]. For example, with the implemented version of this algorithm in Tulip, it takes 9s and 24MB for a complete DAG containing 200 nodes and 19900 edges, 65s and 100MB for a complete DAG with 500 nodes and 124.750 edges [3]. Even if this algorithm was designed to be usable for huge graphs, when we use it on small graphs we observe that the layouts obtained are very similar to the ones obtained by the existing "Sugiyama style layout" algorithms. Figures 8 and 9 show the result of the Buchheim et al [8] algorithm and of our algorithm on a little graph.

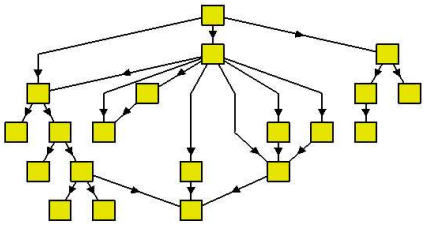

Fig. 8. Our algorithm.

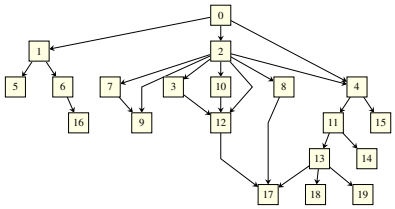

Fig. 9. Buchheim et al algorithm.

3.2 Graph measure

When working with huge graphs, representing them by using graph drawing algorithms is not sufficient to produce a usable view of data for the end-users. Thus, we use intrinsic measures in order to show up the relevant information in graphs. The aim is to give automatically, by using intrinsic parameters, pieces of informations about the data. As for the graph drawing algorithms,

[3] On a x86 computer running at 1.5Ghz.

a wide variety of parameters on graphs exists. For instance, the out-degree of a node. Displaying automatically such pieces of information to the users enables to detect easily, even on huge data-sets, the important pieces of information [20,34].

One of the most important parameters that we use is the Strahler parameter. The Strahler number on binary trees has been introduced in some works about the morphological structure of river networks [28]. It consists in associating an integer value to each of the nodes of a binary tree. These values give a quantitative information about the complexity of each sub-tree of the original tree [33]. Furthermore, if we consider an arithmetical expression A and its evaluation binary tree T, Ershov [11] has proved that the Strahler number of T increased by one is exactly the minimal number of registers needed to compute A. The computation of the Strahler numbers is given by algorithm 1; figure 10 shows an example of the result.

Algorithm 1: Strahler algorithm.

```
binaryStrahler(node η of a binary tree T)
begin
  if η is a leaf of T return 1
  let η_left and η_right be respectively the left and the right child of η
  if binaryStrahler(η_left) is equal to binaryStrahler(η_right)
    return binaryStrahler(η_left) + 1
  else
    return max(binaryStrahler(η_left),binaryStrahler(η_right))
end binaryStrahler
```

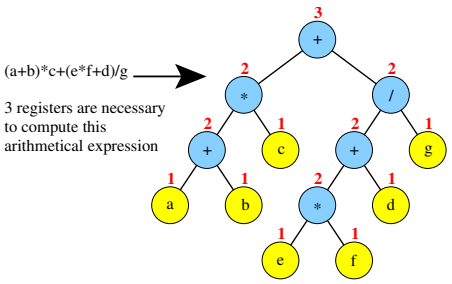

Fig. 10. Arithmetical expression tree.

The algorithm set-up in Tulip [3] is an extension of the Strahler algorithm that can be performed on graphs. The idea is to use the extension of the Strahler number parameter introduced by Fédou [13] and to extend it to graphs. To summarize the algorithm, in the original Strahler version, trees

are seen as arithmetical expressions on which we count the number of registers they need to be evaluated on a computer. When we treat graphs, we consider that a graph can be seen as a program. In this case, to evaluate a program we need registers and nested calls to a stack. Thus, the parameter is in two dimensions : the first one counts the number of registers and second one counts the number of stacks. Such a parameter can be used for different tasks. In Tulip we use it to automatically build the ordering of elements needed in the incremental rendering method (implemented in Tulip) introduced by Wills [36]. The results [3] enable to obtain a good abstraction of the graph drawing during the visual exploration in $O(n)$ for DAG and in $O(n \log(n))$ for general graphs.

3.3 Graph clustering

To enable the visual exploration of a large data-set one must propose automatic clustering algorithms to end-users. These algorithms can be divided into two groups : the intrinsic clustering algorithms and the extrinsic clustering algorithms [19]. In Tulip, different kinds of clustering algorithms, based on the density functions of attributes, enable to manage both cases.

One important graph clustering algorithm available in Tulip is dedicated to trees. The idea is to use the well-known intrinsic parameters on trees in order to cluster them. This algorithm [4] uses node-degree and segment-length (a path in which each node has a degree equal to 2) in trees. Using the combinatoric results on distribution of these two parameters on random trees with fixed degree or/and fixed segment length, we have extracted statistical tables that enable us to test in constant time whether or not a subtree is closed to the distribution. The final algorithm [4] enables to prune a tree in order to reduce its number of elements and to detect "abnormal" phenomena automatically and progressively.

3.4 Visual attributes mapping

In order to treat the problem of huge graphs visualization one must use all the capabilities of the human visual perception system. Thus, size, shape and colors must be used. The problem in this case is to map a parameter (of any kind) into a visual attribute. For instance, if one wants to visualize the out degree of graph's nodes, how should we chose the colors of the elements so that we can obtain an efficient visualization. The straightforward solution is to use linear mapping. In this case, it means using a gradient of colors and then choosing a color of element proportional to its value. Such a method works very well when the distribution of the values is uniform. However, when the distribution is not uniform, in most of the cases this solution enables to detect only few different values.

To solve this problem in Tulip, we are using the color mapping algorithm proposed by Herman et al [17]. This algorithm consists in building an approximation of the probability distribution by computing the frequency histogram of an attribute. Once normalized, this histogram gives a discreet form of the attribute distribution. By accumulating the frequencies along the range of values, it produces a discrete density function that can be used to map an attribute on the colors. Figures 11 and 12 give an example of the differences between a linear mapping and a "distribution mapping". In these figures the colors represent values of the "fission" parameter described by Marshal in [20]. It is clear that in this case, using the "distribution mapping" enables users to detect both the existence of several different values and which values are higher than the others.

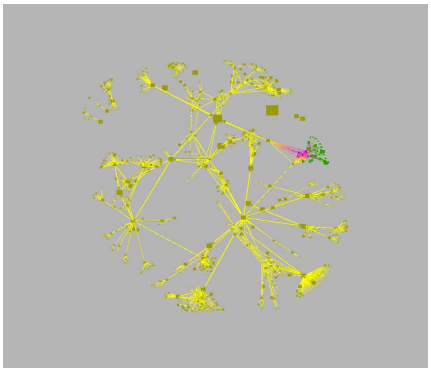

Fig. 11. Linear mapping.

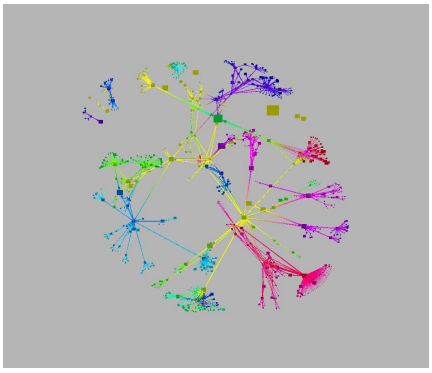

Fig. 12. Herman et al mapping.

4 Implementation

In this section we will describe the data structure used in the Tulip framework. To manage huge graphs, we have put a special emphasis on it. It allows all the operations needed by the Shneiderman exploration process [26] and includes mechanisms that makes it possible to minimize the memory use. The whole Tulip framework has been written in C++. It is based on the standard template library [18] and on a free implementation of the OpenGl library [23]. Then, we will describe the Tulip software [1] that uses the QT [30] library for its dynamic human computer interface, we will also present briefly the TlpRender software that enables to use graphs through a web service. We will conclude by an overview of all the possible extensions of the framework.

4.1 Data structure

Tulip has been built in order to handle graphs having up to 1,000,000 elements (nodes and edges). In this case, the memory management is a crucial factor for building an efficient framework. In order to limit the memory swapping and the processor cache faults, one must put a special emphasis on the trade-off between the memory space and the processor time use. In most of the cases, minimizing the necessary memory implies having a global knowledge of all the stored data. Furthermore, in order to provide a framework for the purpose of information visualization one must treat the problem of attributes storing. Afterwards, we will present the general concepts of the Tulip data structure that are the base of all the Tulip framework.

Graph hierarchy : In Tulip, cluster management is done by using only one graph in the memory and by providing an access to it through views. In order to enable a hierarchical clustering it is also possible to obtain the view of a view. Such a model can be implemented efficiently by using the well-known "flyweight" pattern (for elements) and "chain of responsibility" pattern (for cluster tree) [15]. One of the best advantages is to enable a real sharing of the elements between graphs with a good memory management. This implementation solves the memory problem that appears in the Marshall et al [21] architecture without any reduction of the functionalities.

It is easy to obtain very efficient solutions when using a hierarchy of partitions. However, when working on graphs, which are representing data, the restriction to partitions doesn't enable us to store the clusters of graphs that are already defined in the existing data-set. For instance, when working on the human metabolism data-set, the data are composed of links (edges) between enzymes and compounds (nodes) that form a graph. The set of metabolic pathways is a set of sub-graphs of this graph. In lots of cases, compounds and enzymes can be found in different pathways (for instance, water), thus, this set of sub-graphs cannot be stored if one uses the standard cluster tree definition (cf "[TF2.5]").

The Tulip solution can be considered as graph filtering. In figure 13, one can see a graph G and a set of clusters $\{G_1, G_2, G_3, G_4, G_5\}$. In this example, the clusters are sub-graphs induced by the nodes inside the colored boxes. Figure 14 is the cluster tree that represents the hierarchy of graphs of figure 13. On the tree-edges of figure 14, the boxes represent adjustable filters. The idea of adjustable filters is to store in memory a minimum information in order to rebuild clusters dynamically. For instance, for filter $F2$ it is better to store the difference between G_2 and G if one wants to minimize the memory use. On the contrary, for filter $F3$, it is better to store all the nodes and edges of G_3. For a study describing how adjustable filters can be chosen in order to minimize the memory, and to preserve efficient access to the graph structure, one can refer to [2].

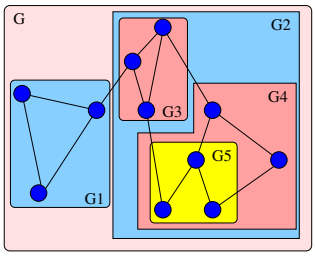

Fig. 13. A clustered graph.

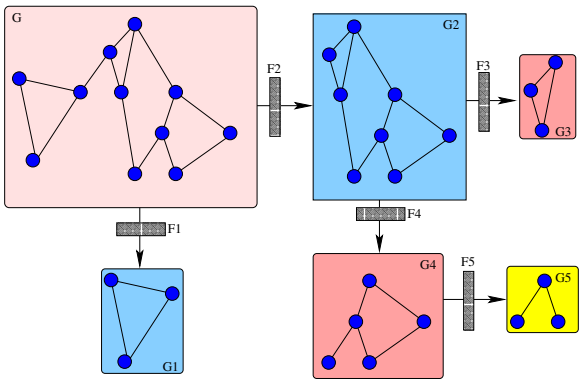

Fig. 14. A cluster hierarchy.

To maintain the coherence of the data structure, one must add some precisions about the modification operations. When one wants to add a node or edge to a cluster, this node or edge must be inserted into all the upper-graphs of the cluster. When one wants to delete a node or an edge in a cluster, this node or this edge must be deleted from all the sub-graphs of the cluster. In practical cases of graph visualization, the processor time-cost due to the management of the hierarchy coherence is offset by the reduction in the memory use and by the improvements in important operations such as graphs cloning or elements sharing.

Graph attributes : An attribute is a value attached to a node or an edge. In a graph visualization framework we can divide the attributes in two types : the ones predefined in the data or set by the user, and the ones computed by the algorithms provided by the framework. For instance, when we load a labeled graph, the nodes' labels are predefined attributes, and when we compute a graph layout, the nodes' coordinates are computed attributes. It is easy to see that in both cases, we can abstract the attributes with a set of two functions that return a value for an element. In Tulip, this abstraction allows the optimization of the memory consumption. For instance, setting to zero all elements of a graph $G(V, E)$ is equivalent to building a

function $F_V : V \rightarrow \{0\}$. It also enables to include useful mechanisms such as buffered evaluation of recursive functions. To include such an improvement we have built functions called proxies which use other functions. Information about proxies can be found in the Design Pattern book written by Gamma et al [15]. Such a mechanism is widely used when we manipulate combinatoric parameters on graphs.

In graph visualization, one must manipulate several attributes at the same time. For example, if one visualizes a filesystem, the names, sizes and types of the files are attributes of the graph nodes. In Tulip, an attribute manager is used to manage a functions set. It allows us to store an unbounded number of attributes and to modify dynamically the set of attributes attached to a graph. For instance, it makes it possible to store several graphs drawings at the same time.

During the clustering, one wants to keep the attributes on the elements. This is equivalent to sharing the attributes between the graphs. For example, in a filesystem, the files names never change even during the visualization of a sub-directory. However, one also needs to be able to locally change the attribute of a cluster. For instance, if a filesystem has been drawn with a tree map algorithm [7], it can be useful to change the drawing of a sub-directory by using the Reingold and Tilford algorithm [24]. In this case, one can look at the structure of the sub-directory instead of at the file sizes. In Tulip, both cases are taken into account by using the attribute manager and the Cluster tree. The mechanism set-up is analogous to those included in object languages that permit the inheritance, redefinition and dynamic change of the object structure [32]. The difference in Tulip is that we allow the inheritance of data. A complete example is presented below.

In figure 15 we can see a set of graphs with attributes. G is a loaded graph that contains two attributes: a layout and labels. G_1 is a spanning directed acyclic graph (DAG) of G, due to the inheritance of the attributes we have directly the layout and the labels of its elements without any cloning of data. G_2 is also a spanning DAG of G that has been redrawn using a DAG dedicated algorithm. In this case, we redefine the layout attribute locally in G_2. Thus, labels are still inherited from G and the layout is contained in G_2. In the graph G_3, we can see that to define the set of inherited attributes, we choose the closest ones in the cluster tree. Therefore, the layout of G_3 is the one of G_2 and not the one of G.

Extension mechanism : To include new features easily, a plug-in mechanism has been built inside Tulip. It works as follows. During the initialization of the Tulip data structure, Tulip's configuration directories are scanned to find new functions (plug-ins). Those found are added to a function factory. Now, when one asks for a graph attribute (or function) named A, if it doesn't exist (locally or inherited) the attribute manager queries the function factory for a function that has the same name. If it exists, after initialization and

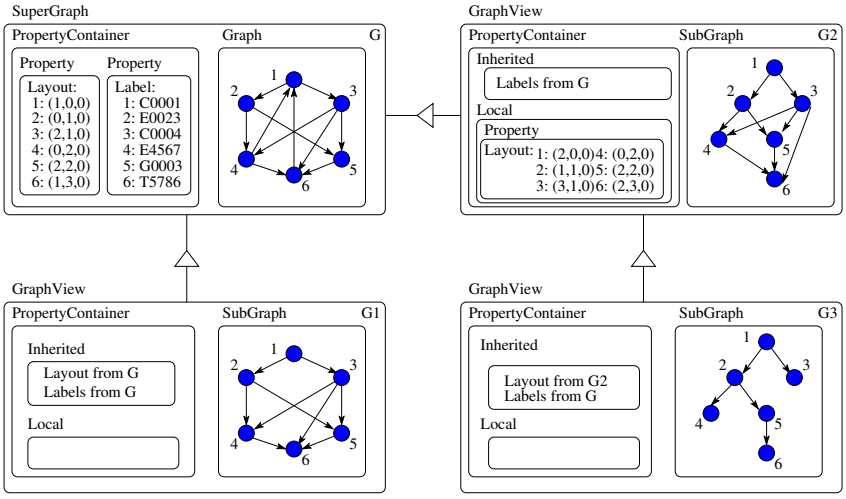

Fig. 15. Inheritance of attributes.

calculability checks, the function (or attribute) is added to the graph. If such a function doesn't exist then a data function is created. Tulip includes other plug-ins mechanism in order to support the addition of algorithms that don't return attributes. The existing ones are the import/export of graphs, the clustering of graphs and glyphs used for the 3D representation of elements.

4.2 Tulip and TlpRender

The first implementation made with the Tulip graph visualization framework is the Tulip software, this software enables to manipulate all the existing plug-ins and attributes available in the framework. The human computer interface of the software is dynamically built according to the plug-ins available in the framework. Thus, it can be seen as a generic software for graphs visualization. Such a software solves the problem of making experimentations with end-users for whom the cost of a specific software building is in most of the cases too important to try research results.

Another software called TlpRender enables to receive a graph in input and then, after applying a graph drawing algorithm (available in the framework) allows to produce a html file in which image maps are used to enable an interaction with the drawn graph. This software is used to make web services that use graphs.

4.3 Overview

Figure 16 presents an overview of the available components in the Tulip package. The first one is the kernel itself that enables to receive all the plug-ins and that manages the data structure introduced above. The second one

is an Open Gl rendering module which enables to display graphs stored in the kernel efficiently. One must notice that the components are made to be reused outside of the Tulip software, the TlpRender software proves it. In figure 16, the green boxes represent all the possibilities of plug-in extensions without any modification and compilation of the existing components. The number of available plug-ins is about 70 and it grows up quickly due to the simplicity of extensions of the framework. One of the best advantage of the framework is that when one adds a new plug-in to the kernel, all the softwares that use the kernel have a direct access to it. Thus, by extending the kernel, we extend all Tulip based on softwares.

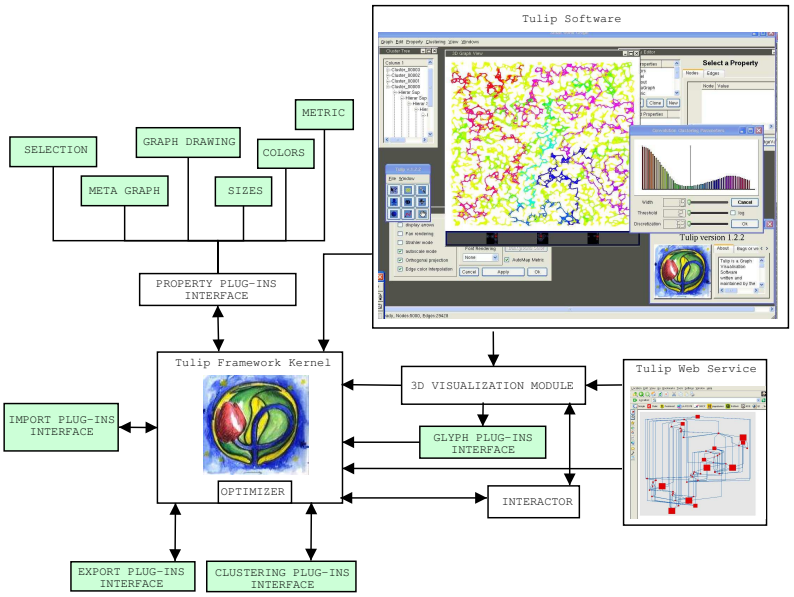

Fig. 16. Tulip Framework overview.

5 Examples

We will give here several examples obtained with the graph drawing algorithms available in the Tulip framework. Figures 17, 18, 19, 20 and 21 show different automatic drawings of a 110.000 files filesystem that can be used in order to detect problems, as for instance, duplication or hidden files. Figures 26, 23, 24 and 25 are different automatic drawings of the LaBRI's web site ($|V| = 800, |E| = 1400$). Figure 22 is a meta-graph that has been built by using the results of Tulip a clustering algorithm on the graph of the inclusions of the Tulip source code. The final graph is the quotient graph (automatically

built) of the subgraphs obtained by the clustering algorithm. On this figure one can see that Tulip can manage Meta-Graph rendering.

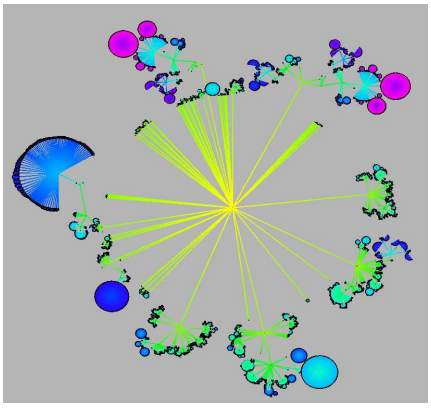

Fig. 17. Balloon drawing ($< 10s$).

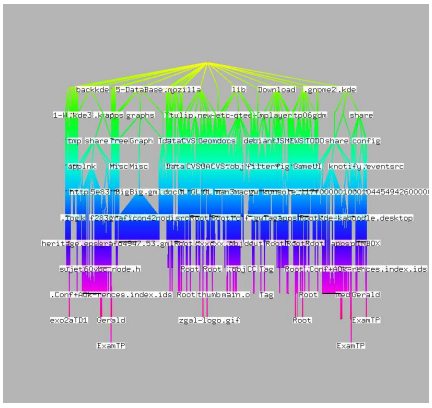

Fig. 18. Hierarchical drawing ($< 1s$).

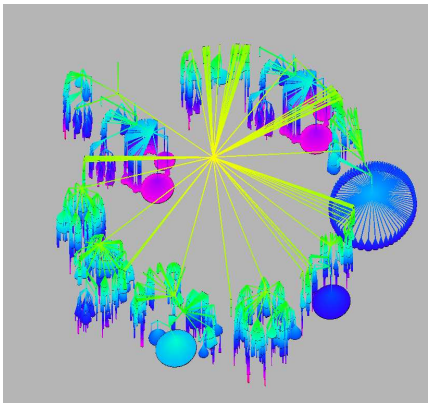

Fig. 19. Cone tree drawing ($< 1s$).

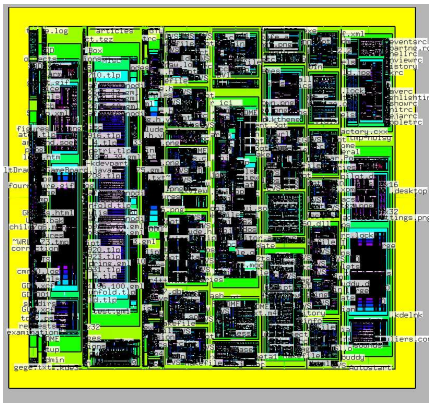

Fig. 20. Tree Map drawing ($< 1s$).

6 Software

The Tulip framework can be downloaded at the following URL: www.tulip-software.org. The whole software is under the general public license (GPL), thus, it can be downloaded and used freely. One of the GPL license restrictions is that each program using the Tulip framework must respect the GPL

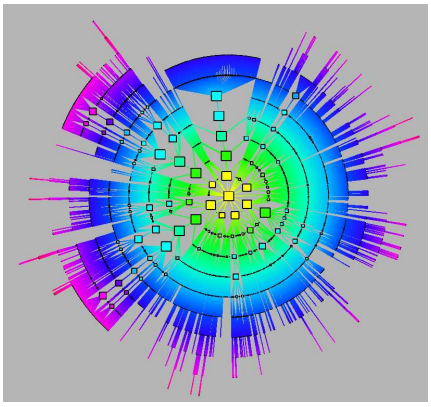

Fig. 21. Radial drawing ($< 1s$).

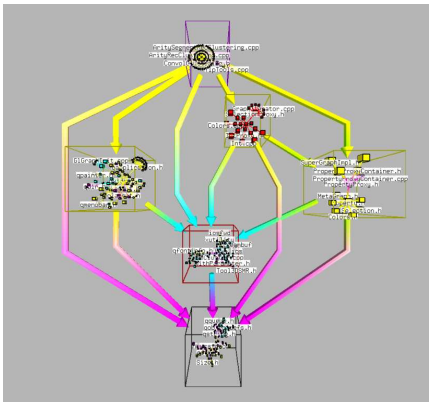

Fig. 22. Meta graph drawing ($< 1s$).

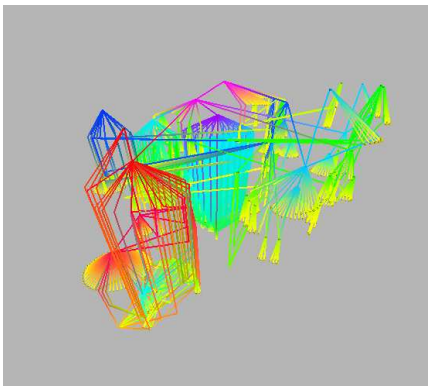

Fig. 23. 3D drawing ($< 1s$).

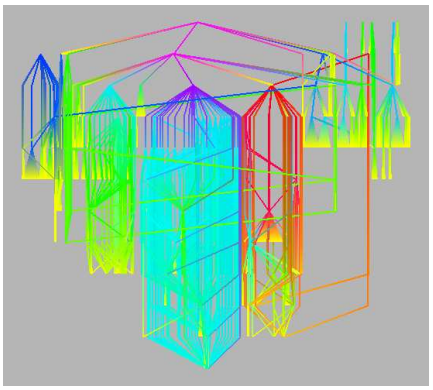

Fig. 24. Hierarchical drawing ($< 1s$).

license. Thus, the project is growing up quickly and new graph drawing algorithm which are implemented using this framework become available to other researchers. The advantage is that it enables us to compare easily graph drawing algorithms and consequently to study efficiently with the end-users the efficiency of each algorithm in order to solve a visualization task.

References

1. D. Auber. Tulip. In Sebastian Leipert Petra Mutzel, Mickael Junger, editor, *9th Symp. Graph Drawing*, Lecture Notes in Computer Science, 2265, pages 335–337. Springer-Verlag, 2001.
2. D. Auber. *Outils de visualisation de larges structures de données.* PhD thesis, University Bordeaux I, December 2002.

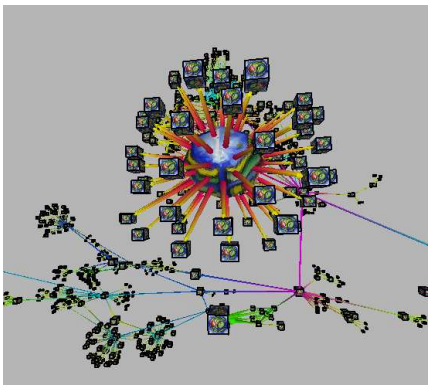

Fig. 25. Spring 3D drawing ($< 90s$).

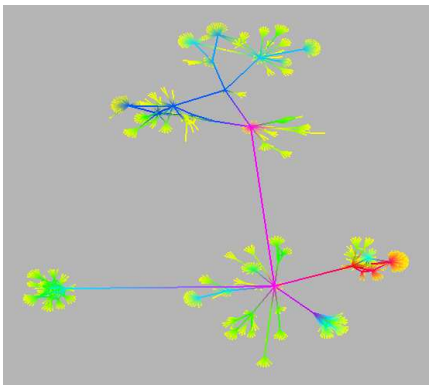

Fig. 26. GEM drawing ($< 12s$).

3. D. Auber. Using strahler numbers for real time visual exploration of huge graphs. In *International Conference on Computer Vision and Graphics*, pages 56–69, september 2002.
4. D. Auber and M. Delest. A clustering algorithm for huge trees. *Advances in Applied Mathematics*, To appear 2002.
5. J. Bertin. *La graphique et le traitement graphique de l'information.* Flammarion, 1977.
6. B. Le Blanc, D. Dion, D. Auber, and G. Melançon. Constitution et visualisation de deux réseaux d'associations verbales. In *Colloque Agents Logiciels, Coopération, Apprentissage et Activité Humaine (ALCAA)*, pages 37–43, 2001.
7. D.M. Bruls, C. Huizing, and J.J. VanWijk. Squarified treemaps. In *Data Visualization 2000*, proceedings of the joint Eurographics and IEEE TCVG Symposium on Visualization, pages 33–42. Springer, 2000.
8. C. Buchheim, M. Junger, and S. Leipert. A fast layout algorothm for k-level graphs. In Joe Marks, editor, *Proc. 8th Symp. Graph Drawing*, Lecture Notes in Computer Science, 1984, pages 229–240. Springer-Verlag, 2000.
9. J. Carriere and R. Kazman. Interacting with huge hierarchies: Beyond cone trees. In G. and S. Eick, editors, *IEEE Symposium on Information Visualization*, pages 74–78. Atlanta, Georgia Institute for Electrical and Electronics Engineers, 1995.
10. A. Dix and J. Finlay. *Human-Computer Interaction.* Prentice-Hall, 1998.
11. A.P. Ershov. On programming of arithmetic operations. *Communication of the A.C.M*, 1(8):3–6, 1958.
12. K.M. Fairchild, S.E. Poltrock, and G.W. Furnas. Semnet: Three-dimensional graphic representation of large knowledge bases. In R. Guindon and L. Erlbaum, editors, *Cognitive Science and its Applications for Human-Computer Interaction*, pages 201–233, 1998.
13. J.M. Fédou. Nombre de strahler sur les arbres généraux. In GDR ALP, editor, *Ecole jeunes chercheur en algorithmique et calcul formel*, may 1999.
14. A.K. Frick, H. Mehldau, and A. Ludwig. A fast adaptive layout algorithm for undirected graphs. In I.G. Tollis R. Tamassia, editor, *2nd Symp. Graph*

Drawing, Lecture Notes in Computer Science, 894, pages 388–403. Springer-Verlag, 1994.
15. E. Gamma, R. Helm, R. Johnson, J. Vlissides, and G. Booch. *Design Patterns.* Addison-Wesley Pub Co, 1995.
16. B. Gärter. Fast and robust smallest enclosing balls. In Springer Verlag, editor, *Algorithms-ESA'99: 7^{th} Annual European Symposium Proceedings, Lecture Notes In Computer Science*, number 1643, pages 325–338, 1999.
17. I. Herman, M. Marshall, and G. Mélançon. Density functions for visual attributes and effective partitioning in graph visualization. In *IEEE Symposium on Information Visualization*, pages 49–56. IEEE Computer Society, 2000.
18. Hewlett-Packard. Stl standard template library. www.sgi.com/tech/stl/.
19. M. Kauffman and D. Wagner. *Drawing Graphs: Methods and Models.* Lecture Notes in Computer Science, 2025. Springer-Verlag, 2001.
20. S. Marshall. *Methods and tools for the visualization and navigation of graphs.* PhD thesis, University Bordeaux I, June 2001.
21. S. Marshall, I. Herman, and G. Melançon. An object-oriented design for graph visualization. *Software Practice and Experience*, 31(8):739–756, 2001.
22. N. Megiddo. Linear-time algorithms for linear programming in R^3 and related problems. In *SIAM J. Comput.*, number 12, pages 759–776, 1983.
23. B. Paul. The mesa 3-d graphics library. www.mesa3d.org.
24. E.M. Reingold and J.S. Tilford. Tidier drawings of trees. *IEEE Transactions on Software Engineering*, 7(2):223–228, 1981.
25. G.G. Robertson, JD. Mackinlay, and S.K. Card. Cone trees: Animated 3d visualizations of hierarchical information. In *SIGCHI, Conference on Human Factors in Computing Systems*, pages 189–194. ACM, 1991.
26. B. Shneiderman. The eyes have it: A task by data type taxonomy for information visualization. In Boulder, editor, *IEEE Conference on visual languages*, pages 336–343, 1996.
27. R. Spence. *Information Visualization.* Addison-Wesley, 2001.
28. A.N. Strahler. Hypsomic analysis of erosional topography. *Bulletin Geological Society of America 63 1117-1142.*, 1952.
29. K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical system structures. *IEEE Trans. Systems, Man and Cybernetics*, SMC-11(2):109–125, 1981.
30. Trolltech. Qt the crossplatform c++ gui framework. www.trolltech.com.
31. W.T. Tutte. How to draw a graph. *Proc. London Math. Soc.*, 3(13):743–768, 1963.
32. D. Ungar, C. Chambers, B.W. Chang, and U. Hölzle. Organizing programs without classes. *Lisp and Symbolic Computation*, 4(3):223–242, july 1991.
33. G. Viennot. Trees everywhere. In A. Arnold, editor, *Colloquium on Trees in Algebra and Programming*, Lecture Notes in Computer Science 431, pages 18–41. Springer-Verlag, 1990.
34. C. Ware. *Information Visualization: Perception for design.* Interactive Technologies. Moragn Kaufmann, 2000.
35. E. Welzl. Smallest enclosing disks (balls and ellipsoids). In Hermann A. Maurer, editor, *New Results and New Trends in Computer Science*, Lecture Notes in Computer Science, 555. Springer-Verlag, 1991.
36. G.J. Wills. NicheWorks : Interactive visualization of very large graphs. In Giuseppe Di Battista, editor, *5th Symp. Graph Drawing*, Lecture Notes in Computer Science, 1353, pages 403–414. Springer-Verlag, 1997.