18

GDToolkit

18.1 Introduction 571
18.2 Key Features of GDToolkit 572
18.3 Graph-classes and their Hierarchy 573
Topology level • Shape Level • Metrics Level
18.4 Constructors 583
18.5 Management of Constraints 585
Topology Constraints • Shape Constraints • Metrics Constraints
18.6 Examples of Applications 589
Internet Analysis • Web Searching • Database Analysis
Acknowledgements 594
References 595

Giuseppe Di Battista
University "Roma Tre"

Walter Didimo
University of Perugia

18.1 Introduction

GDToolkit (available at http://www.dia.uniroma3.it/∼gdt) is an object-oriented graph drawing library, written in the C++ programming language. It provides many facilities that support users to develop specific graph visualization interfaces that can be used in real-world domains.

The computation of a drawing is typically decomposed into a sequence of logical steps, and several algorithms can be chosen for each step, which offer different compromises in terms of efficiency and effectiveness. Developers can tune the ratio between the performance of their applications and the quality of the computed drawings, by combining the different algorithms available for each step. Generic drawing algorithms and drawing conventions can be customized and tailored for a specific application context by means of different types of constraints that the developer can apply on the drawings.

The design of GDToolkit started in 1996, as a part of the ALCOM-IT European Project. For the use of basic data-structures like vectors, lists, maps, and sets, GDToolkit was originally strongly based on the LEDA library [MN95, MN00]. After several years, the current version of GDToolkit is now completely LEDA free, since the basic data-structures have been totally re-implemented. GDToolkit is now under a commercial license; detailed information about license terms and conditions can be found at the official web page of the project.

This chapter describes the main functionalities and architectural aspects of GDToolkit and it is structured as follows. The key features and the design principles of GDToolkit are first described (Section 18.2). Specific architectural aspects concerned with the design and the use of class constructors are then examined (Section 18.4). The constraint management

system of the library is discussed in Section 18.5. Finally, some examples of real-world applications developed using GDToolkit are illustrated (Section 18.6).

18.2 Key Features of GDToolkit

Several key features have been taken into account in the design of GDToolkit. They are listed and discussed below:

A specific class for each type of graph. In GDToolkit each type of graph is modeled as a specific class, called a *graph-class*. Graph-classes are organized into a hierarchy that reflects different levels of abstraction, ranging from graph topology to graph geometry (see Section 18.3 for a detailed description of the hierarchy). A similar architecture has been previously proposed in other projects of graph drawing libraries [BBDL91, DGST90]. There are basic graph-classes to model graphs with different topological properties, like general multi-graphs, directed graphs, planar graphs, flow networks, trees. In addition, there are graph-classes for representing graphs with associated some drawing information. For example, there exist intermediate graph-classes that model orthogonal drawings and upward drawings only in terms of drawing "shape" (see, e.g., [DETT99]), and there are graph-classes that model drawings of graphs in terms of vertex and edge-bend coordinates.

All the graph algorithms implemented in GDToolkit are encapsulated as methods of the topmost graph-class in which they are safely applicable. Derived graph-classes inherit methods from the ancestor ones, optionally refining or hiding them when unsafe. Inheritance and encapsulation effectively help the application developer in dealing with the intrinsic complexity of graph algorithms and data-structures.

A graph drawing algorithm is viewed as a sequence of steps. Each step maps an object of a graph-class to an object of another graph-class. A drawing is typically the result of a sequence of constructors; each time a constructor is applied to a graph-object g, a new graph-object g' is created and equipped with additional drawing features with respect to g. For example, the code in Figure 18.1 shows how to create an orthogonal drawing of a graph as a simple sequence of constructors.

```
/* creates a graph ug, loading it from file "my-graph" */
undi_graph ug;
ug.read ("my-graph");
/* computes a planar embedding for ug, with possible crossing nodes */
plan_undi_graph pug (ug);
/* computes an orthogonal shape for the planar embedded graph */
orth_plan_undi_graph opug (pug);
/* compacts the orthogonal shape to create the final drawing */
draw_undi_graph dug (opug);
```

Figure 18.1 A fragment of code that computes an orthogonal drawing of the graph described by the graph-object ug. The graph is loaded from a file and the drawing is computed according to the topology-shape-metrics approach [Tam87]. Each computation step is performed by a different constructor.

Efficient object constructors. Suppose that B is a graph-class that inherits from A. Invoking a constructor of B that takes in input a graph-object g of A has the effect of creating a new graph-object g' of B that contains additional drawing information (attributes) with respect to g. Typically, in the construction process, all the structures of g are first copied in the state of g', and then the state of g' is enriched with additional data computed by some algorithms. Sometimes however, once g' has been created, g is no longer needed in the program. In these cases one may wish to "promote" g to become an object of class B, avoiding to duplicate data for the structures of g. Such a promoting mechanism makes it possible to save computational time and space resources, especially when g describes a large graph. The graph-classes of GDToolkit are designed to allow that. Details about the promoting mechanisms of GDToolkit are given in Section 18.4.

Management of Constraints. As many other graph drawing libraries, GDToolkit is not devoted to a specific application field. It is mainly thought as a general purpose graph drawing collection of objects and algorithms, which can be used in several real-world contexts. However, different application domains may need to deal with different variants of a generic graph drawing convention, depending on the specificity of the domain itself. These variants often reflect into a set of drawing constraints, and therefore it is crucial that the drawing algorithm is able to deal with these constraints. For example, some applications might require that a subset of edges is not allowed to cross, or that some vertices should have a prescribed dimension.
GDToolkit makes it possible to customize its drawing conventions and its drawing algorithms by means of an effective constraint management system. The graph-objects of GDToolkit can be equipped with constraints that can be viewed as additional properties for vertices, edges, or faces. Constraints can be added or removed at each time of the life-cycle of a graph-object. If a constraint is added to a graph-object, this constraint remains consistent even if the object is updated. Also, GDToolkit constructors automatically preserve (and in case enforce) the constraints when a new graph-object is created as a refinement or as a copy of an existing graph-object. The constraint management system of GDToolkit is described in detail in Section 18.5

Extensibility. In order to make the extensibility of the library easy, the definition of new classes, constructors, and constraints is done according to specific patterns, which should be taken into account by programmers that wish to extend the library. The principles of these patterns are described in Sections 18.3, 18.4, and 18.5.

In the next section the architecture of GDToolkit is described, focusing on the key features discussed above. Several code and drawing examples are provided in order to better illustrate the use of the library.

18.3 Graph-classes and their Hierarchy

The graph-classes of GDToolkit are structured into a hierarchy, and provide objects for each specific type of graph. The design of the hierarchy is mainly driven by the well-known *topology-shape-metrics approach* [DETT99, Tam87] for orthogonal drawings. According to this approach, a drawing is computed into three phases:

Topology: A *planar embedding* of the input graph is computed, by possibly adding dummy vertices to replace crossings if the graph is not planar; the planar embedding is described by the circular lists of edges incident to each vertex, or equivalently by the set of faces.

Shape: An *orthogonal shape* is computed within the planar embedding found in the previous phase, where one of the faces is chosen as the external face; the shape describes the sequence of left and right bends along the edges, and the angles formed by two consecutive edges incident to the same vertex in a circular order.

Metrics: The final position of the vertices and bends is computed, while preserving the shape determined in the previous phase. Then, dummy vertices are removed.

GDToolkit applies and extends this approach to other drawing conventions. Indeed, more in general, a drawing is described (and constructed) at three different levels of abstractions, where each level adds drawing information to the parent level. The first level describes the topology (embedding) of the drawing, the second level its "shape", and the third level the final geometry of the drawing in terms of vertex and edge-bend coordinates. The shape of a drawing can be regarded as a partial description of the drawing that typically determines the relative position of vertices and edges, without deciding their final placement. For some drawing conventions the concept of shape does not make sense, and in this case it is possible to skip an abstraction level in the construction of the drawing. The hierarchy of the main graph-classes of GDToolkit is depicted in Figure 18.2.

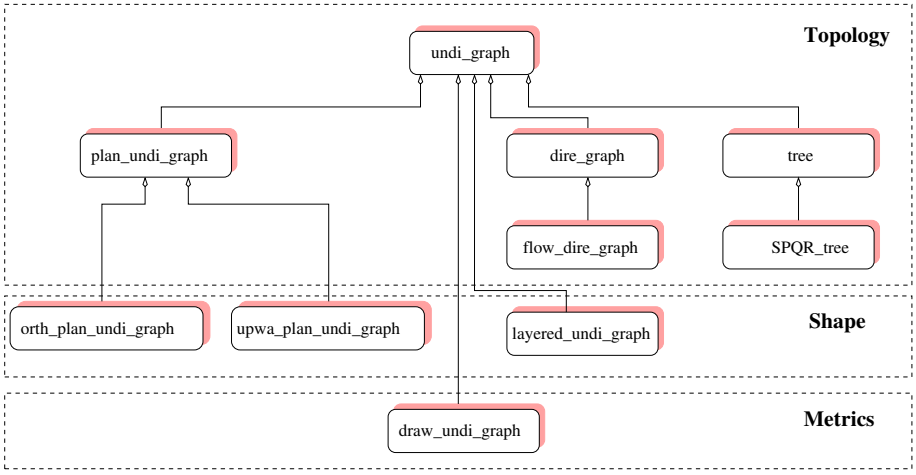


Figure 18.2 The hierarchy of the main graph-classes of GDToolkit.

18.3.1 Topology level

The root of the graph classes hierarchy is the *undi_graph* class, whose objects represent generic graphs that can be connected or not, and that can have multiple edges and self-loops. Also, any edge of an *undi_graph* object can be optionally oriented, i.e, an *undi_graph* can have both undirected and directed edges at the same time.

Each node and each edge of an *undi_graph* object is associated with a non-negative integer identifier. No duplication of identifiers is allowed in the same class of elements. Methods for automatically generating identifiers are provided by the library, but identifiers can also be

manually set or changed by the programmer. When a graph object is copied into another graph object, nodes and edges are duplicated while preserving their identifiers. Therefore, identifiers can be used to keep a one-to-one correspondence between the set of nodes and edges of the two graphs.

An *undi_graph* stores information about its embedding, i.e., the circular ordering of the edges incident to every node. This embedding is preserved during any copy operation of the graph. Also, class *undi_graph* contains a large set of basic methods to access and update the topology of the graph, and advanced methods to deal with its embedding, orientation, and connectivity. For example, there are methods that test the existence of planar embeddings for the graph, or of planar *bimodal embeddings* in the case the graph has only directed edges. We recall that a bimodal embedding for a planar digraph is such that, for each vertex, all the incoming edges (as well as all the outgoing edges) are consecutive around the vertex. There are methods to compute *st*-orientations, methods to connect the graph by adding a minimal set of extra edges, and methods to perform different kinds of traversal of the graph. Figure 18.3 shows a fragment of code that creates a new *undi_graph* object `ug` reading the structure of the graph from a file, executes two copies `ug1` and `ug2` of `ug`, and updates (if possible) the embedding of `ug1` into a planar one, and the embedding of `ug2` into a planar bimodal one, after an orientation for `ug2` is found. Note that, if `ug2` is biconnected, the program computes an *st*-orientation for it.

```
/* creates an object ug, loading it from file "my_graph",
 * and makes two copies of ug */
undi_graph ug;
ug.read("my_graph");
undi_graph ug1 (ug);
undi_graph ug2 (ug);

/* makes the embedding of ug1 planar, if possible*/
if (!ug1.make_embedding_planar ())
    cout << "\nThe graph is not planar" << flush;

/* if graph ug2 is biconnected, makes it st-oriented,
 * else makes it randomly oriented*/
if (ug2.is_biconnected())
     ug2.make_directed(ug2.first_node(),ug2.last_node());
else ug2.make_directed(true);

/* makes the embedding of ug2 planar bimodal, if possible */
if (!ug2.make_embedding_cand_planar())
    cout << "\nThe oriented graph is not planar bimodal" << flush;
```

Figure 18.3 A fragment of code that illustrates how to use some methods of the *undi_graph* class.

Most graph algorithms implemented as methods of an undi_graph object runs in linear time. For example, an *st*-orientation of a graph with 200,000 vertices and 600,000 edges is executed in about 14 seconds under Linux on a typical machine with i5-540M Intel processor and 4 GB RAM.

Embedded planar graphs are modeled by the class *plan_undi_graph*, which enriches the basic topological structure of an *undi_graph* with the description of a set of faces. Following the philosophy of the library, a *plan_undi_graph* object can be created using a constructor that takes as a constant parameter an *undi_graph* object. This constructor applies a planarity testing algorithm and, if the graph is not planar, a planarization algorithm that replaces crossings with "dummy" nodes, called *crossing nodes*.

The planarity testing algorithm implemented in GDToolkit is the one described by Boyer et al. [BCPD04]; it has been shown to be faster than the one implemented in the LEDA library [MN95, MN00]. The planarization algorithm is based on a technique that inserts an edge per time by following a shortest path in the dual graph of the planar embedded graph computed so far [DETT99]. While planarizing sparse graphs is rather fast, executing the planarization algorithm on dense graphs might require a significant computational effort, due to the high number of crossings. For example, a graph with 500 vertices and 750 edges is planarized in about 13 seconds under Linux on a computer with i5-540M Intel processor and 4 GB RAM. A much smaller but much denser graph consisting of 100 vertices and 500 edges is planarized in about 42 seconds.

Class *tree* offers methods to perform standard operations on ordered rooted trees, like visits in different orders, re-rooting, and so on. The *SPQR_tree* class inherits from class `tree`, and models the data structure introduced by Di Battista and Tamassia [DT96] to represent the triconnected components and the different embeddings of a biconnected graph. It is possible to use *SPQR*-tree objects to enumerate and change the embeddings of a graph, although the current implementation of SPQR-trees in GDToolkit only deals with planar graphs. In GDToolkit, *SPQR*-trees are extensively used to implement branch-and-bound algorithms that compute drawing with the minimum number of bends in the variable embedding setting [BDD00, BDD02].

Class *dire_graph* defines specialized methods performing on directed graphs. A special subclass of *dire_graph* is the *flow_dire_graph* class, which represents flow networks [AMO93] with *capacities*, *costs*, and *flow values* for the arcs. Every node of a flow network may also have a certain *balance* value that can be either negative or positive, depending on the fact that the node demands or supplies flow (by default, the balance value of a node is zero, which means that its total entering flow equals its total leaving flow). The class provides methods to compute feasible flows in a given network while optimizing some function, like for example the total cost. Several drawing algorithms in the library extensively use a *flow_dire_graph* object to compute a feasible flow with prescribed value and minimum cost. The flow is computed in a network that is typically constructed from the topology of the graph to be drawn.

18.3.2 Shape Level

At the shape level, GDToolkit offers three classes: *orth_plan_undi_graph*, which models orthogonal representations; *upwa_plan_undi_graph*, which models upward and quasi-upward representations, and *layered_undi_graph*, which models layered graphs. In the following we give some details about orthogonal, upward, and quasi-upward representations in GDToolkit, by also recalling some basic concepts related to these drawing conventions.

Orthogonal representations in GDToolkit are modeled according to the *simple-podevsnef* drawing convention[BDD00], a simplified and pretty robust version of the *podevsnef* convention (also called *Kandinsky*) defined in [FK96]. Vertices are represented as small rectangles, all having the same size, and any vertex can have any number of incident edges.

The library offers different algorithmic choices to compute orthogonal representations, with different compromises between efficiency and effectiveness. Figure 18.4 shows two

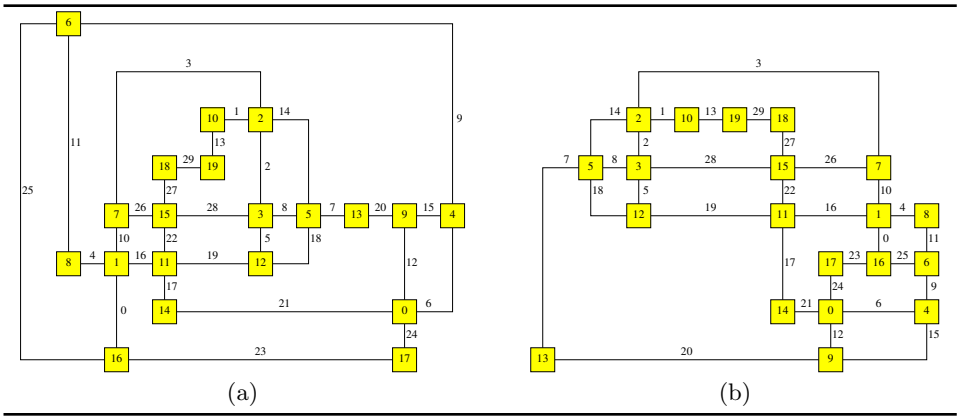


Figure 18.4 Two orthogonal representations of the same planar graph computed by GDToolkit. (a) An orthogonal representation with the minimum number of bends for a given planar embedding; (b) An orthogonal representation with the minimum number of bends over all possible planar embeddings of the graph. Node and edge identifiers are shown in the drawing.

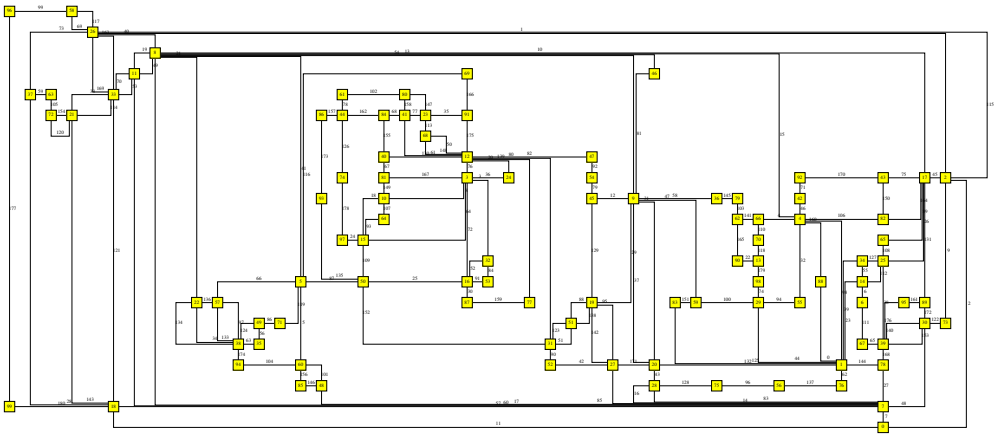


Figure 18.5 An orthogonal representation of a graph with 100 vertices.

examples of orthogonal representations of a planar graph G, one having the minimum number of bends within the given planar embedding of G, and the other having the minimum number of bends over all planar embeddings of G. The representation in (a) has been computed with an $O(n^2 \log n)$-time algorithm based on a flow technique that extends the one described in [Tam87]. The representation in (b) has been computed with an exponential-time algorithm based on a branch-and-bound technique, which enumerates and explores the embeddings of the graph using $SPQR$-trees. Both the polynomial-time algorithm and the exponential-time algorithm are described in [BDD00].Figure 18.5 shows an orthogonal representation of a graph with 100 vertices. The computation of a minimum-bend orthogonal drawing for an embedded planar graph with 100 vertices and 200 edges takes about 0.2 seconds under Linux on a machine with i5-540M Intel processor and 4 GB RAM. Computing a bend-minimum orthogonal drawing over all planar embeddings for a graph with 30 vertices and 50 edges takes about 10 seconds.

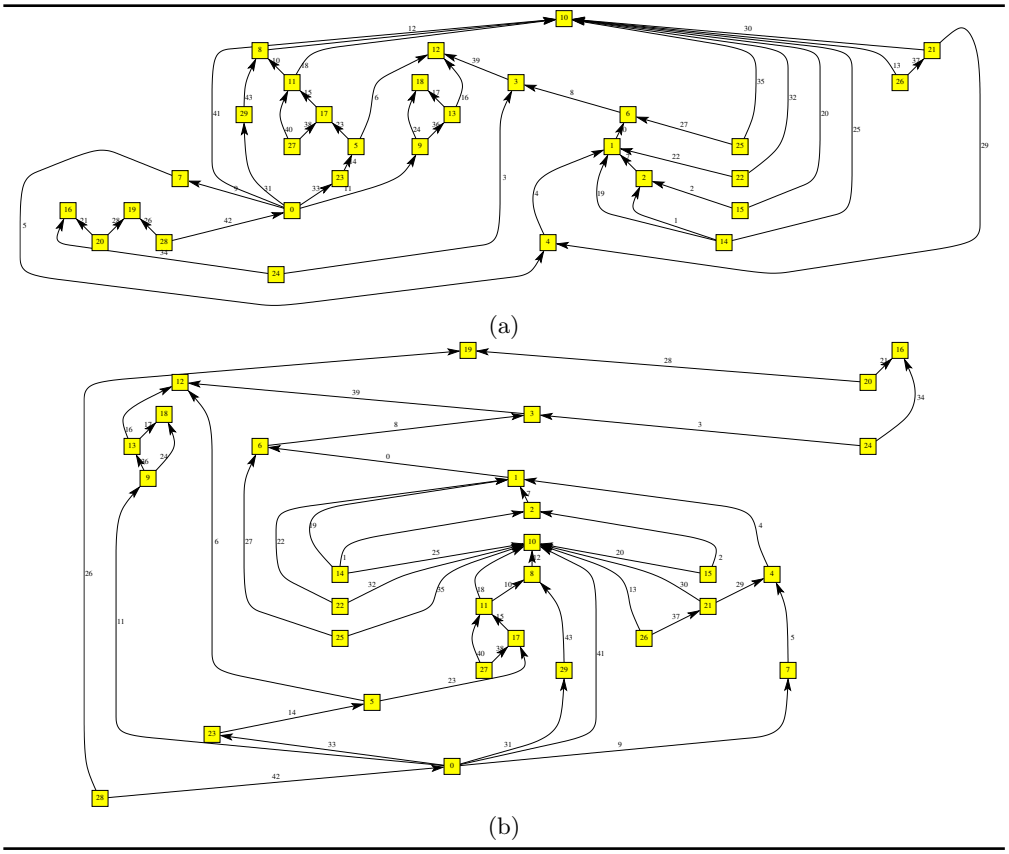


Figure 18.6 Two quasi-upward planar representations of the same digraph, computed by GDToolkit. (a) A representation with two bends on edge 5 and two bends on edge 29; (b) A representation with no bend, i.e., it is an upward planar representation.

Concerning upward representations, GDToolkit adopts the *quasi-upward* drawing convention defined by Bertolazzi et al. [BDD02]. We recall that an *upward drawing* of a directed graph is a drawing such that each vertex is represented as a distinct point of the plane and each edge is drawn as a simple curve monotonically increasing in the upward direction (i.e., from bottom to top), according to its orientation. An upward planar drawing is a drawing that is planar and upward a the same time. An upward planar drawing can exist only if the digraph is acyclic and admits a bimodal embedding. An *upward planar representation* is a partial description of an upward planar drawing, which defines the two linear lists of outgoing and incoming edges for each vertex, without fixing the final positions of the vertices. Unfortunately, acyclicity and bimodality are not sufficient conditions for the existence of an upward planar drawing, and in practice most digraphs do not admit such a layout. A quasi-upward drawing is a generalization of an upward drawing, which allows *bends* along the edges. A bend is a point in which the edge inverts its vertical direction, switching from upward to downward or vice-versa (if the edge is drawn as a smoothed curve, a bend along the edge is a point with horizontal tangent for the edge). The only requirement of a quasi-upward drawing is that for each directed edge (u, v), the edge enters v from below and leaves u from above. This implies that each edge has an even number of bends (possibly

zero bends). Every digraph admits a quasi-upward drawing (even if it is acyclic) and a planar digraph admits a quasi-upward planar drawing if and only if it admits a bimodal embedding. Note that, an upward drawing can be regarded as a quasi-upward drawing with no bends along the edges. A *quasi-upward planar representation* is a partial description of a quasi-upward planar drawing; it defines the two linear lists of incoming and outgoing edges for each vertex and the sequence of bends along the edges.

As for orthogonal representations, GDToolkit provides different methods to compute a quasi-upward planar representation of a digraph. Figure 18.6 shows two examples of quasi-upward planar representations of the same digraph; the first representation is computed by using a flow-based $O(n^2 \log n)$-time algorithm that minimizes the number of bends within a given planar bimodal embedding; the second one is computed with a branch-and-bound exponential-time algorithm that minimizes the number of bends over all planar bimodal embeddings of the digraph. The algorithms for computing quasi-upward representations are those described in [BDD02]. In practice, the computation of a quasi-planar representation is very fast and takes less time then computing orthogonal drawings. For example, a quasi-planar representation of a bimodal planar digraph with 200 vertices and 240 edges is computed in about 0.05 seconds under Linux on a computer with i5-540M Intel processor and 4 GB RAM.

```
/* creates a graph ug, loading it from file "my-graph" */
undi_graph ug;
ug.read("my_graph");
/* computes a planar embedding for ug, with possible crossing nodes */
plan_undi_graph pug (ug);
/* computes an orthogonal shape for the planar embedded graph,
 * specifying the external face and the desired algorithm */
orth_plan_undi_graph opug (pug,pug.last_face(),PLAN_ORTH_OPTIMAL);
```

Figure 18.7 A fragment of code that creates an orthogonal shape of a graph.

GDToolkit also offers the possibility of orienting an undirected embedded planar graph in such a way that the number of its sources and sinks is minimized and it has an upward planar representation. As described in [DP03], this helps in the implementation of drawing algorithms for visibility representations in case the graph is not biconnected. Observe that, for a biconnected graph an upward orientation with the minimum number of sources and sinks always coincides with an *st*-orientation of the graph.

Objects of classes *orth_plan_undi_graph* and *upwa_plan_undi_graph* are usually constructed from *plan_undi_graph* objects, by specifying the wanted layout algorithm. It is also possible to specify an external face if the selected algorithm preserves the planar embedding. To give an example, consider the simple code in Figure 18.7. It constructs an *orth_plan_undi_graph* object `opug` by the *plan_undi_graph* object `pug`. When `opug`'s constructor is invoked, a face of `pug` is chosen to be the external face; if such a face is not specified, it is chosen as the first in the list of faces of `pug`. The algorithm `PLAN_ORTH_OPTIMAL` selected to construct `opug` corresponds to the algorithm that computes an orthogonal representation of the graph in the simple-podevsnef model, with the minimum number of bends within the given planar embedding.

18.3.3 Metrics Level

At the bottom level of the hierarchy GDToolkit provides the *draw_undi_graph* class, which is very easy to use. Indeed, an object of this class is an *undi_graph* object with additional basic geometric information, like vertex-coordinates and bend-coordinates; *draw_undi_graph* objects are also equipped with some attributes to define colors and labels for vertices and edges.

The basic philosophy of the *draw_undi_graph* class is to provide one or more constructors from each other graph-class of the library. Often, GDToolkit provides different algorithms to compute a drawing in a specific convention; each algorithm has a different trade-off between drawing aesthetics and time performance. For instance, an orthogonal drawing can be computed from an *orth_plan_undi_graph* object by selecting an algorithm in a wide set of compaction algorithms, obtained by combining different alternatives like:

- Decomposing the faces of the orthogonal representation into rectangles [Tam87] or into *regular faces* [BBD+00].
- Computing the coordinates of vertices and bends with a linear-time algorithm based on topological numbering or with an $O(n^2 \log n)$-time algorithm based on flow-techniques [DETT99].
- Applying or not a one-dimensional compaction post-processing to further reduce the area and the total edge length of the drawing, if possible.

The code in Figure 18.8 computes two different orthogonal drawings with the same shape. The first drawing, `dug1`, is computed by applying the fastest compaction algorithm in the library, while the second one, `dug2`, is constructed by using the slowest compaction algorithm. The resulting drawings, `dug1` and `dug2`, are depicted in Figure 18.9; observe that `dug2` is much more compact in terms of area and total edge length.

```
/* creates a graph ug, loading it from file "my-graph" */
undi_graph ug;
ug.read("my_graph");
/* computes a planar embedding for ug, with possible crossing nodes */
plan_undi_graph pug (ug);
/* computes an orthogonal shape for the embedded graph */
orth_plan_undi_graph opug (pug);

/* computes two drawings of the orthogonal shape,
 * using different compaction algorithms */
draw_undi_graph dug1 (opug, FAST_COMPACTION);
draw_undi_graph dug2 (opug, SLOW_REGULAR_COMPACTION_2_REFINED);
```

Figure 18.8 A fragment of code that computes two different orthogonal drawings with the same shape.

As another example, visibility and polyline drawings can be directly computed from an object `pug` of class *plan_undi_graph*, by choosing between a linear-time compaction algorithm or a polynomial-time compaction algorithm based on flow techniques [Did00]. Indeed, for these kind of drawing conventions the concept of shape is not defined. Figure 18.10 shows two visibility drawings and two polyline drawings of the same embedded planar graph. The

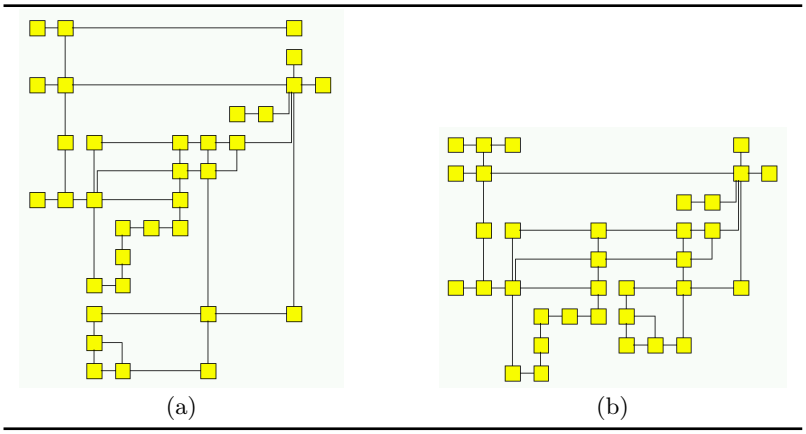


Figure 18.9 (a) Drawing `dug1` and (b) drawing `dug2`, computed with the code of Figure 18.8. The two drawings have the same shape but different geometry. The drawing in (b) is much more compact, both in terms of area and in terms of total edge length.

drawings in Figures 18.10 (a)-(b) are obtained by executing a linear-time drawing algorithm, while the drawings in Figures 18.10 (c)-(d) are obtained by applying an $O(n^2 \log n)$-time compaction algorithm that reduces the total edge length.

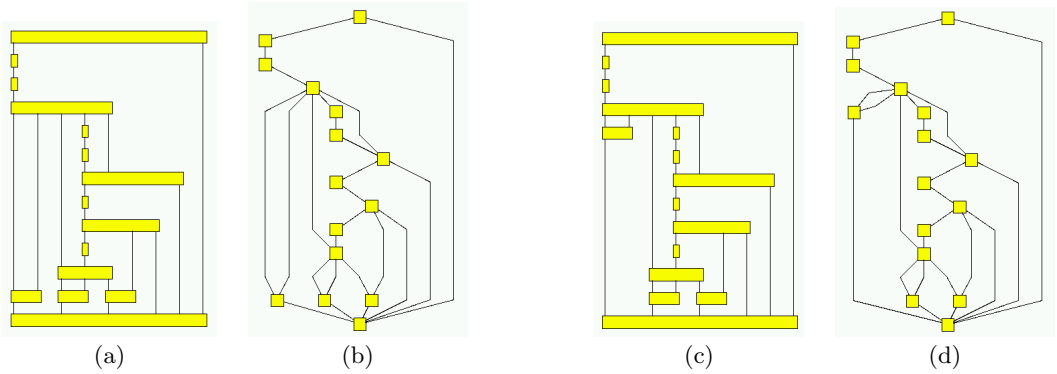


Figure 18.10 Two visibility drawings and two polyline drawings of the same embedded planar graph. The total edge length of the drawings (c) and (d) is smaller than the one of the drawing (a) and (b).

It is also interesting to observe that, since a quasi-upward drawing can be computed by using a visibility representation as intermediate step, the same compaction algorithms applied above can be used for computing a quasi-upward drawing of a quasi-upward represen-

tation. For example, Figure 18.11 shows two different drawings of the same quasi-upward planar representation, computed with different compaction algorithms. The drawing in Figure 18.11 (b) has smaller total edge length. GDToolkit also implements a recent algorithm [Did05, Did06] for compacting upward planar representations, which is based on the concept of *switch-regular* faces, introduced in [DL98]. According to this strategy, the augmentation of the upward planar representation to an including *st*-digraph is not performed by using the face decomposition described in [BDLM94], but it is done by first decomposing the faces into switch-regular ones. This avoids the insertion of useless extra edges and typically leads to drawings that have better aspect ratio (see Figure 18.12).

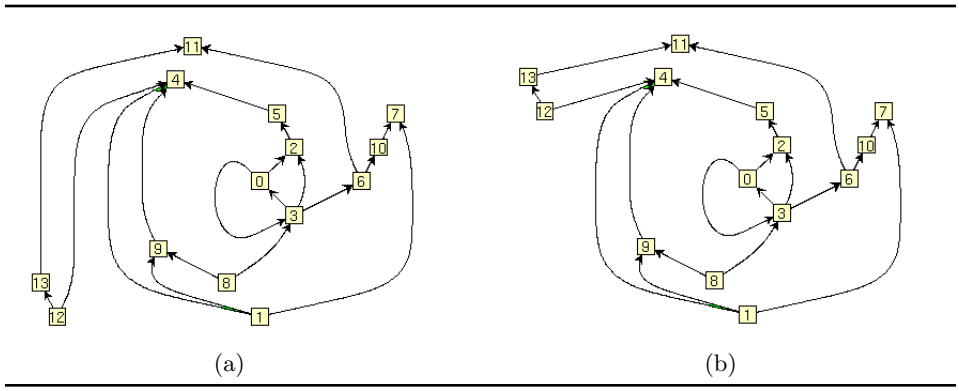


Figure 18.11 Two different quasi-upward drawings of the same quasi-upward representation. The total edge length of the drawing in (b) is smaller that the total edge length of the drawing in (a).

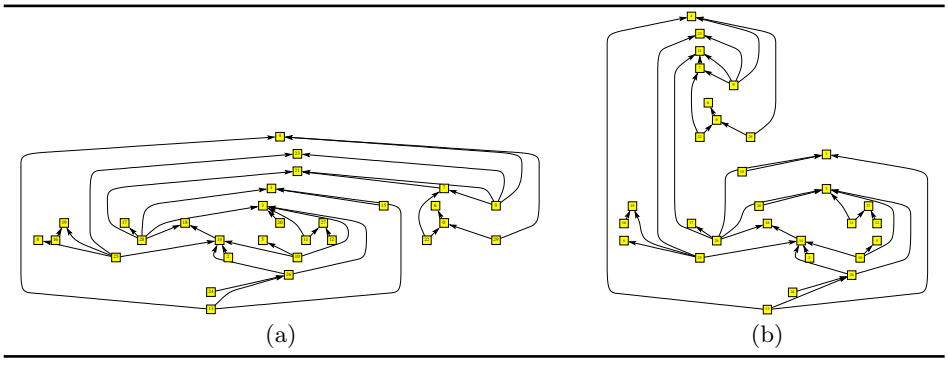


Figure 18.12 Two upward drawings of the same upward representation. (a) The drawing has been computed using the standard augmentation technique described in [BDLM94]. (b) The drawing has been computed with the new algorithm described in [Did05].

18.4 Constructors

As observed in the previous sections, a drawing algorithm in GDToolkit typically reflects in a path of constructors. For this reason, constructors play a crucial role in the library and they are written following a common pattern, which is depicted in Figure 18.13.

Suppose that a graph-class B inherits a graph-class A. According to the pattern of Figure 18.13, a constructor of B first invokes a constructor or a copy operator of A to transfer the inherited information; then, the constructor of B invokes a private method, `local_new`, that allocates memory for the local structures that are needed to store additional information, and finally it calls another private method, `loca_init`, that computes and stores the data in the new local structures.

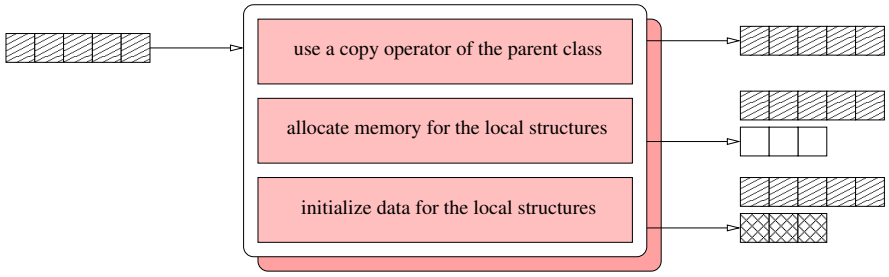


Figure 18.13 A schematic illustration of the design pattern of GDToolkit constructors.

As a concrete example, imagine that a *plan_undi_graph* object `pug` is created from an *undi_graph* object `ug`. Object `pug` must have the same nodes and edges as `ug` and, additionally, it defines a set of faces and possible extra nodes that replace crossings. The construction of the planar embedding of `pug` is done by applying a planarization algorithm on the topology of `ug`, possibly subject to some planarization constraints (see Section 18.5). Figure 18.14 shows the code of a constructor of class *plan_undi_graph*. Parameter `po` specifies if the new graph object must have the same embedding as `ug` or not. Parameter `err_mess` enables/disables an error-handler in the case some planarization constraints can not be satisfied. Method `local_new` allocates memory for the list of faces, while method `local_init` executes the planarization algorithm.

```
    plan_undi_graph::
plan_undi_graph (const undi_graph& ug, planarize_options po, bool err_mess)
  {
  /* copies the basic structure of the graph (nodes and edges) */
  undi_graph::operator=(ug);
  /* creates the additional data structures required by
   * a plan_undi_graph object */
  local_new();
  /* executes a planarization algorithm to computes faces
   * and related objects */
  local_init(po,err_mess);
  }
```

Figure 18.14 A constructor of class *plan_undi_graph*. The code reflects the pattern illustrated in Figure 18.13.

As mentioned in Section 18.2, another key aspect of GDToolkit is the possibility of constructing a new graph-object by means of a *promoting* mechanism. Suppose for example that a is an *undi_graph* object and suppose we want to construct a *plan_undi_graph* object b with the same set of vertices and edges as a. As explained above, b enriches the information stored in a with a set of faces, which defines a planar embedding for a. Suppose also that a is no longer needed in the program after the construction of b; indeed, b contains a super-set of information of a. In this situation it could be useful to get b as the result of a promoting procedure applied to a that avoids duplication of data, so saving computational time and memory space. The graph-classes of GDToolkit support such a promoting mechanism by means of a public method, called `steal_from`. Referring to the example above, method `steal_from` invoked on b "steals" the data-structures of a and then constructs a set of new data-structures to store the additional information of b (in the specific example a set of faces). To make this idea efficient, the instance variables in the graph-classes of GDToolkit are just references (pointers) to the data-structures that contain the data. This implies that b can steal the data of a by simply copying in constant time the internal references of a. After this operation, both a and b link the same data-structures, and therefore update collisions may happen. To avoid this drawback, method `steal_from` automatically cleans the references of a, which becomes as an "empty" object. Figure 18.15 shows a schematic description of the promoting mechanism. Figure 18.16 gives an example of use of method `steal_from`.

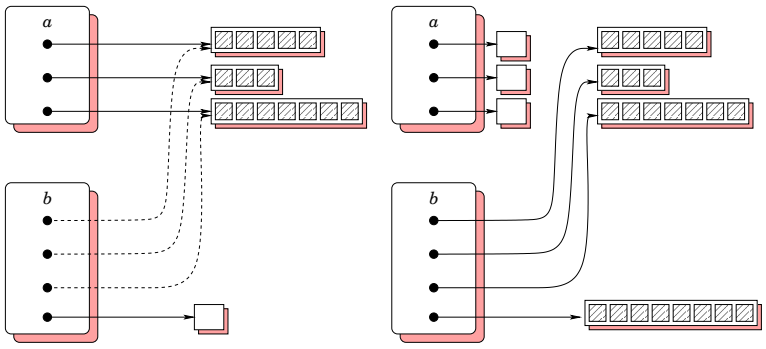


Figure 18.15 A schematic description of the promoting mechanism. Object b is the result of the promoting, and object a is made useless after the promoting process.

```
/* creats an undi_graph object (*ug), loading it from file "my-graph" */
undi_graph *ug = new undi_graph();
(*ug).read ("my-graph");
/* computes an empty planar embedded graph */
plan_undi_graph pug();
/* initializes pug with the nodes and edges of (*ug);
 * object (*ug) will be useless from now on */
pug.steal_from(*ug);
/* (*ug) is deallocated from the main memory */
delete(ug);
```

Figure 18.16 A *plan_undi_graph* object is constructed promoting an *undi_graph* object.

18.5 Management of Constraints

GDToolkit is equipped with a flexible architecture for managing constraints. Different types of constraint can be concurrently applied on the graph, which are taken into account by the involved layout algorithms. A *constraint* type is a reference to an object as well as types *node* and *edge*, and each constraint still has a unique identifier. The *undi_graph* class provides a set of methods for adding, removing, and copying constraints.

Constraints in GDToolkit have a special "intelligent" management system, which is explained in the following points.

- Each constraint is described by a specific set of parameters that depends on the type of the constraint itself. For example, a constraint that makes an edge e not crossable is described by the only parameter e; a constraint that forces a vertex v to have height h and width w is described by the triple (v, h, w). In addition, each constraint has an internal read-only parameter that specifies the type of constraint. This type can be accessed by means of a public method.
- A graph G' that is obtained as a copy or by inheritance of a graph G, also inherits all constraints of G. Furthermore, constraints react according to their type to all the relevant events occurring on their node and edge parameters. More precisely, each type of constraint is represented by a specific class that encapsulates its behavior with respect to changes of the nodes and edges involved in the constraint. An abstract class provides the set of virtual reaction methods common to all the derived constraint classes, and each derived constraint class provides its own implementation for each reaction method.
- The main events with a potential impact on a constraint applied on a given node/edge are the deletion, the split, and the merge of that node/edge. For each of these events, each constraint class defines a reacting method. For example, if an edge e is split into two edges e_1 and e_2, a reaction method called `update_after_edge_split()` is automatically invoked on all the constraints applied on e, so that each constraint executes its specific implementation of this method.
- Theoretically, any number of constraints can be set on a graph at any time. However, each algorithm decides its own policy about each kind of constraint. This means that the programmer can decide to implement an algorithm that takes into account or not a specific type of constraint. Also, some constraints could be not compatible to each other; in this case, an algorithm that takes them into account, typically causes an error.

GDToolkit currently offers several types of predefined constraints involving both topology, shape, and metrics. The use of constraints in the topology-shape-metrics framework have been addressed in several papers, including [BDLN05, CGM+10, DDLP10, EFK00, GKM08, Tam98]. GDToolkit implements some of the constraints described in the literature or their variants. However, any programmer can define a new constraint by extending the base abstract class and by providing an implementation for each reaction method. In the following the main predefined constraints of GDToolkit are described.

18.5.1 Topology Constraints

Concerning the topology of a graph, GDToolkit provides three different types of constraints; all of them are taken into account by the planarization algorithm.

The first type of constraint imposes that an edge e is not allowed to cross any other edge. If edge e is split into two edges e_1 and e_2, the constraint is propagated on both e_1 and e_2. Symmetrically, if an edge e is obtained by merging two edges e_1, e_2, and at least one of them is not crossable, then e will become not crossable too. If the planarization algorithm encounters an edge e that is not crossable, it omits to insert its dual edge in the dual graph of the planar embedded graph computed so far. This implies that a shortest path in the dual graph never intersects e.

The second type of constraint imposes that a specified set of vertices $\{v_1, v_2, \dots v_k\}$ belongs to the same face. In order to maintain this property, the planarization algorithm temporarily adds to the graph a *star gadget*, consisting of a dummy vertex u and dummy edges $(u, v_1), (u, v_2), \dots, (u, v_k)$, where the dummy edges of the star are made not crossable, applying on them the previous type of constraint. The star gadget is removed at the end of the planarization process. Figure 18.17 shows an example of application of this constraint.

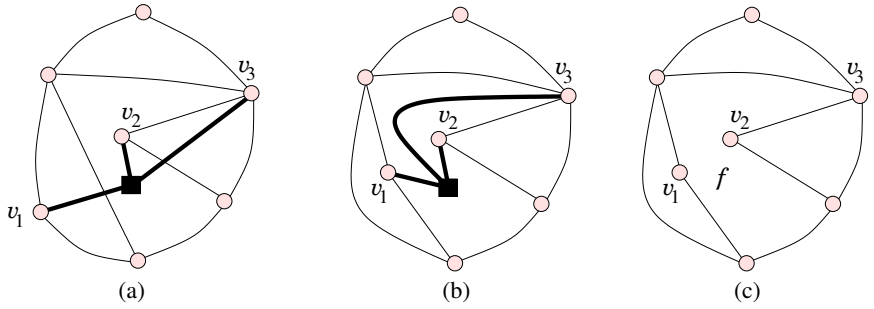


Figure 18.17 Illustration of the star gadget used to force a set of vertices to stay in the same face. In this example, the vertices are v_1, v_2, v_3. (a) A star gadget is added; it consists of the square black vertex and of the bold edges. (b) A planar embedding of the enhanced graph is computed; (c) The final planar embedding for the input graph. At the end of the planarization process, v_1, v_2, v_3 belong to the same face f.

The third type of topological constraint is a variant of the previous one. It imposes that a certain set of vertices $\{v_1, v_2, \dots, v_k\}$ belongs to the same face f and that these vertices circularly occur on the boundary of f in the specified order. To satisfy this constraint, the planarization algorithm uses the star gadget shown above, with the additional property that the circular sequence of edges incident to the dummy vertex of the star gadget is fixed.

Figure 18.18 shows two orthogonal drawings: The drawing in (a) has been obtained without any topological constraint. The drawing in (b) has been computed imposing that vertices $7, 12, 1, 0$ belong to the same face (the face is highlighted). In order to satisfy this constraint, the planarization algorithm introduced some edge-crossings.

18.5.2 Shape Constraints

At the shape level, GDToolkit provides several predefined constraints that are taken into account by the flow-based algorithms that compute orthogonal and quasi-upward representations. These constraints are listed and discussed below.

- *Number of bends per edge.* This constraint can be applied on an edge e, in order to establish a certain policy in bending e. Two different policies can be applied:

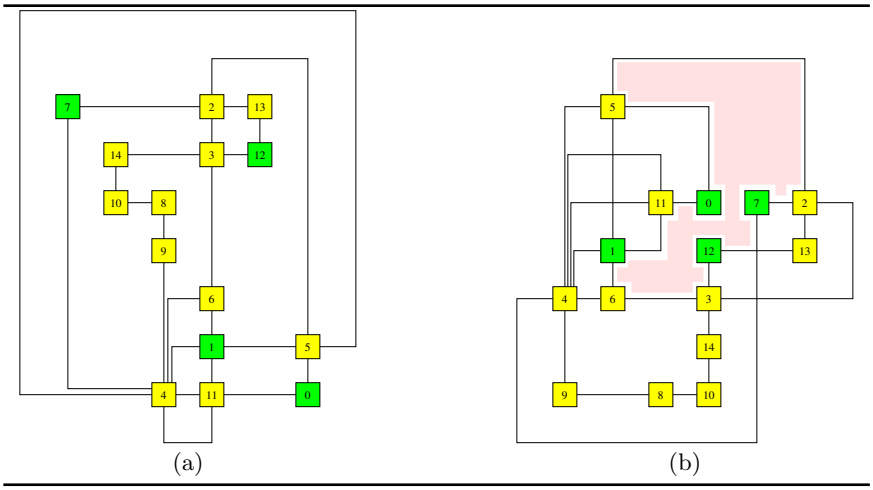


Figure 18.18 Two orthogonal drawings of the same graph. (a) The drawing has been computed with no constraint. (b) The drawing has been computed forcing vertices $7, 12, 1, 0$ to stay in the same face.

  - Edge e must have zero bends, i.e., it must be a straight-line edge.
  - Edge e can have any number of bends. This means that the algorithm will assign a zero cost to each bend of e, and therefore e will turn any time this avoids to bend other edges.

  Each of the two policies is translated into a suitable constraint in the flow network associated with the orthogonal or quasi-upward representation. We recall that in such a flow network (see also [BDD02, DETT99, Tam87]) there is a node v_f for each face f of the graph and there is a pair of directed arcs $e_{fg} = (v_f, v_g), e_{gf} = (v_g, v_f)$ for each edge e shared by two (possibly coincident) faces f and g. The flow along the arcs e_{fg}, e_{gf} determines the right bends and the left bends along e in the final representation. In order to guarantee that e has no bend in the representation, it is sufficient to set an infinite cost (or zero upper capacity) on e_{fg}, e_{gf}. Conversely, in order to assign the highest turn priority to e, one can assign cost zero and infinite upper capacity to both e_{fg} and e_{gf}.

- *Turn direction.* This constraint forces an edge $e = (u, v)$ to turn only in a specified direction. This means that e can be forced to have only right bends or only left bends, while moving on it either from u or from v. To implement this constraint, we just remove in the flow network one of the two arcs e_{fg}, e_{gf}, where f and g are the two (possibly coincident) faces shared by e.
- *Angle type.* This last constraint allows the programmer to decide the value that a specified angle must have in an orthogonal representation. Possible angle values in degrees are $\{0, 90, 180, 270, 360\}$. The constraint is specified by a triple (e, v, a), where e is an edge incident to v, and a is the value of the angle formed at v between edge e and its successive edge in clockwise order around v. This type of constraint is still translated into a suitable constraint in the flow network associated with the orthogonal representation. More precisely, it is sufficient to fix the value of the flow along the arc of the network that connects v to the face in which the

angle lies; indeed, this flow value defines the value of the angle in the orthogonal representation.

Figure 18.19 shows an example of use of shape constraints.

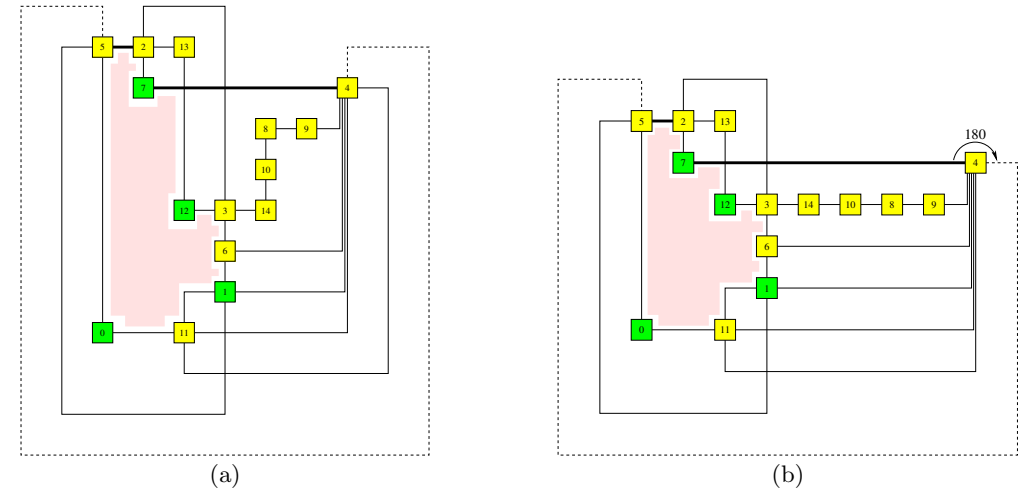


Figure 18.19 (a) The orthogonal drawing has been computed from the graph of Figure 18.18, with the constraint that edges $(5,2),(4,7)$ cannot bend (the edges are in bold), while edge $(5,2)$ can have any number of bends (the edge is dashed). (b) An orthogonal drawing computed by adding the further constraint that the angle at vertex 4 to the right of edge $(4,7)$ is a 180 degrees angle.

18.5.3 Metrics Constraints

Concerning the metrics of a drawing, GDToolkit currently offers two predefined constraints for orthogonal drawings.

The first constraint allows the programmer to customize the size of each vertex, independently to each other. More in details, every vertex v can be drawn has a rectangle having a predefined width w and a predefined height h in terms of units of an integer coordinate grid. In absence of constraints, v will be drawn as a small rectangle that occupies only a grid unit, that is, v will have width and height equal to zero. The constraint on the dimension of the nodes is handled in the compaction step of the topology-shape-metrics approach, by using the flow-based technique described in [DDPP99]. Figure 18.20 shows two orthogonal drawings of the same embedded planar graph. In the drawing of Figure 18.20 (a) all vertices have dimensions zero, while in the drawing of Figure 18.20 (b) some vertices have been expanded imposing constraint dimensions. Observe that the shape of the two drawings is the same.

The second constraint makes it possible to specify the points where an edge will be incident to a side of a vertex. More precisely, assume that an edge e is incident to a vertex v. An orthogonal representation fixes the side s of v on which e will be incident. If on v a

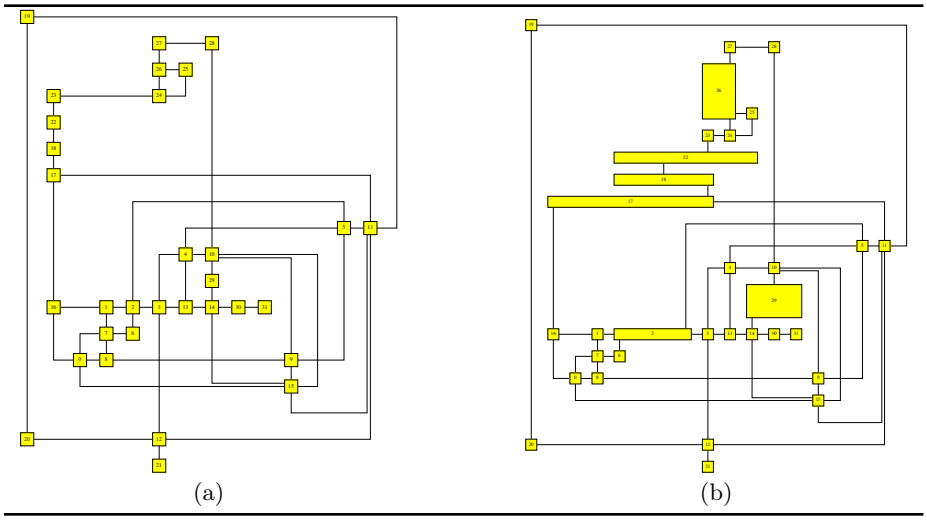


Figure 18.20 Two orthogonal drawings with the same shape: (a) The drawing has no constraint; (b) The dimensions of some vertices have been preassigned.

dimension constraint has been fixed so that s has length l, the programmer can impose any distance $d \leq l$ between the incidence point of e on s and a corner of s (see Figure 18.21).

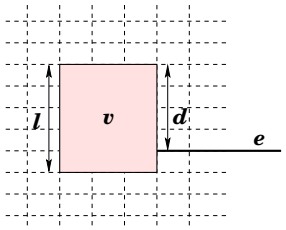


Figure 18.21 Illustration of the constraint that makes it possible to fix the incidence point of an edge on the side of a vertex in an orthogonal drawing.

18.6 Examples of Applications

The GDToolkit library has been effectively used to develop several applications in different real-world domains, which is a proof of its flexibility. In the following we briefly discuss some of these applications.

18.6.1 Internet Analysis

At a high level of abstraction, the Internet can be seen as a network of so called *Autonomous Systems*. An Autonomous System (AS in the following) is a group of sub-networks under the same administrative authority, and is identified by a unique integer number. In this sense, an AS can be seen as a portion of the Internet, and the Internet can be seen as the

totality of the ASes. To maintain the reachability of any portion of the Internet, each AS exchanges routing information with a subset of other ASes, mainly selected on the basis of economic and social policies. To exchange information, the ASes adopt a routing protocol called *BGP* (Border Gateway Protocol). This protocol is based on a distributed architecture where *border routers* that belong to distinct ASes exchange information about the routes they know. Two border routers that directly communicate are said to perform a *peering session*, and the ASes they belong to are said to be *adjacent*. The *ASes graph* is the graph having a vertex for each AS and one edge between each pair of adjacent ASes. The ASes graph consists of more than $10,000$ vertices and then it is not reasonable to visualize it completely on a computer screen.

Internet Service Providers are often interested in visualizing and analyzing the structure of the ASes graph and the related connection policies, in order to extract valuable information on the position of their partners and competitors, capture recurrent patterns in the Internet traffic, and detect routing instabilities. Several tools have been designed for this purpose (see, e.g., [DK01] for references). The system `Hermes` [CDD+02] is based on the GDToolkit facilities, and allows users to incrementally explore the Internet topology by means of automatically computed maps. The basic graph drawing convention used to represent the maps is the Kandinsky model for orthogonal drawings. However, since the handled graphs often have many vertices of degree one connected to the same vertex, the Kandinsky model is enriched with new features for effectively representing such vertices.

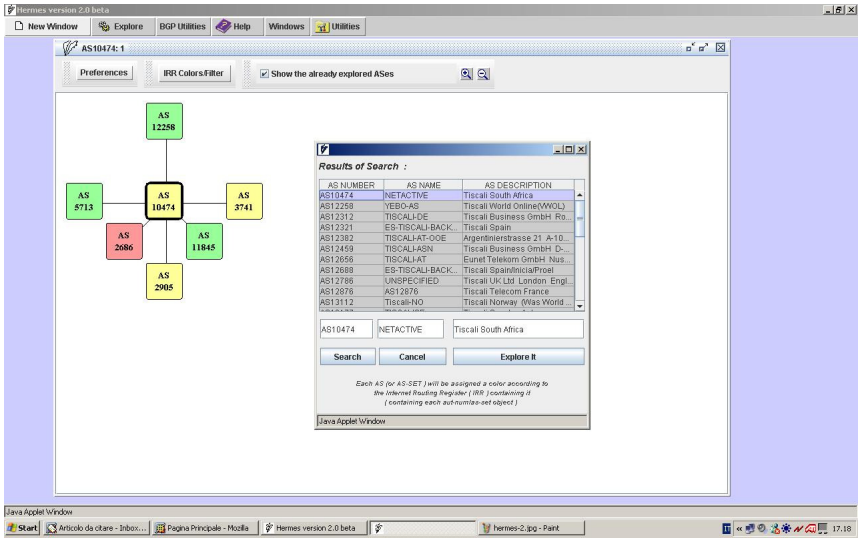
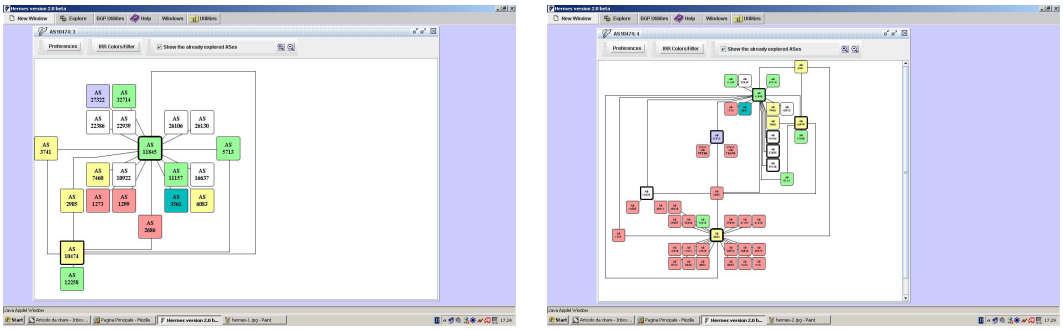


Figure 18.22 A map showing the ASes adjacent to AS10474, NETACTIVE, Tiscali South Africa. (Figure taken from [DL07].)

The graphical user interface of `Hermes` offers several exploration facilities. The user can search for a specific AS and can start the exploration of the Internet from that AS. At each successive step, the user can display information about the routing policies of the ASes contained in the current map, or she can expand the map by exploring one of these ASes. For example, Figure 18.22 shows a snapshot of the system where the AS10474 (NETACTIVE, Tiscali South Africa) is searched and selected by the user for exploration; a first map that consists of the ASes adjacent to AS10474 is then automatically computed



(a) (b)

Figure 18.23 (a) A new map obtained from the previous map by exploring AS11845. (b) A more complex map obtained by performing several exploration steps. (Figure taken from [DL07].)

and displayed by the system. Figure 18.23 shows how the map is expanded when the user decides to explore other ASes.

18.6.2 Web Searching

The output of a classical Web search engine consists of an ordered list of links (URLs) that are selected and ranked according to the user's query, the documents content, and (in some cases, like `Google`) the popularity of the links in the World Wide Web. The returned list can however consist of several hundreds of URLs and users may omit to check some URLs that might be relevant for them just because these links do not appear in the first positions of the list.

A *Web meta-search clustering engine* is a system conceived to support the user in retrieving data from the Web by overcoming some of the limitations of traditional search engines. A Web meta-search clustering engine provides a visual interface to the user who submits a query; it forwards the query to (one or more) traditional search engines, and returns a set of clusters, also called *categories*, which are typically organized into a hierarchy. Each category contains URLs of documents that are semantically related to each other and is labeled with a string that describes its content. As a consequence, the user of a meta-search clustering engine has a global view of the different semantic areas involved by her query and can more easily retrieve the Web data relative to those topics in which she is interested.

Although an effective representation of the categories and of their semantic relationships is essential for efficiently retrieving the wanted information, most Web meta-search clustering engines (see, e.g., `Vivísimo`, `iBoogie`[1], `SnakeT`[2] [FG04, FG05]) have a GUI in which the hierarchy of clusters is displayed as a tree. However, this type of representation may not

[1] http://www.iboogie.com/
[2] http://snaket.di.unipi.it/

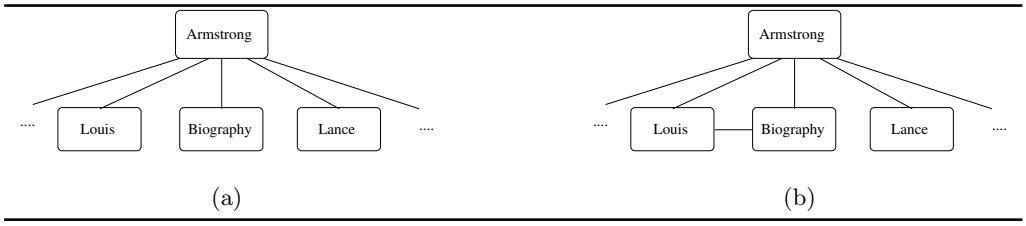


Figure 18.24 (a) A portion of a tree of categories for the query "Armstrong". (b) The tree is equipped with an edge that highlights cluster relationships.

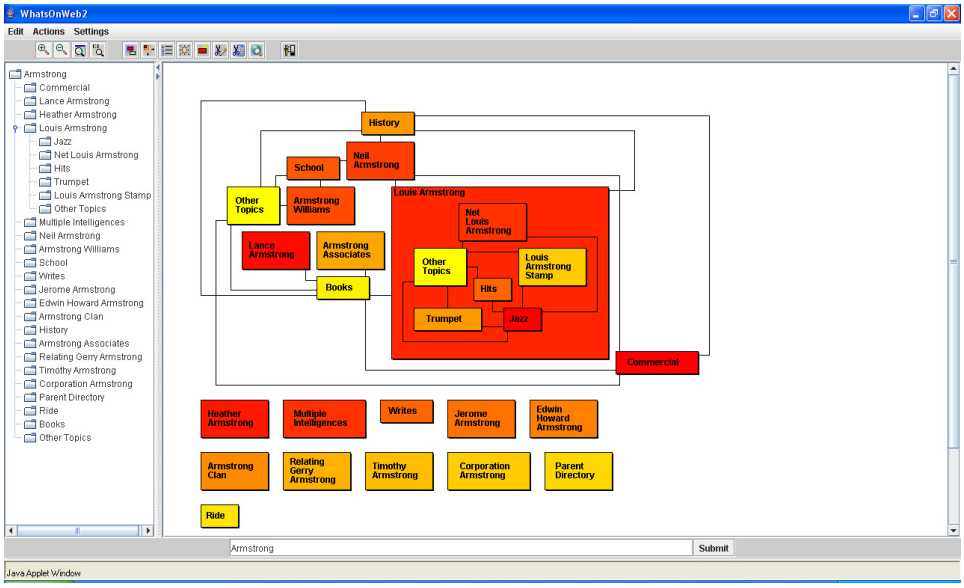
be fully satisfactory for a complex analysis of the returned Web data. Suppose for example that the user's query is "Armstrong" and that the clusters hierarchy returned by a Web meta-search clustering engine is the tree depicted in Figure 18.24 (a). Is the category "Biography" related to "Louis" or to "Lance" or to both (or to no one of them but to the astronaut Neil Armstrong?). If instead of a tree, the systems returned a graph as the one in Figure 18.24 (b), the user would be facilitated in deciding whether the category "Biography" is of her interest.

`WhatsOnWeb` [DDGL05, DDGL06, DDGL07] is a meta-search clustering engine that makes it possible to retrieve data from the Web by using drawings of graphs. The nodes represent categories of semantically coherent URLs and the edges describe relationships between pairs of categories. The graphical environment of `WhatsOnWeb` consists of two frames (see, e.g., Figure 18.25). In the left hand side frame the hierarchy of categories is represented as a classical directories tree. In the right hand side frame the user interacts with the drawing of a *clustered graph*[FCE95], where each cluster coincides with a semantic category.

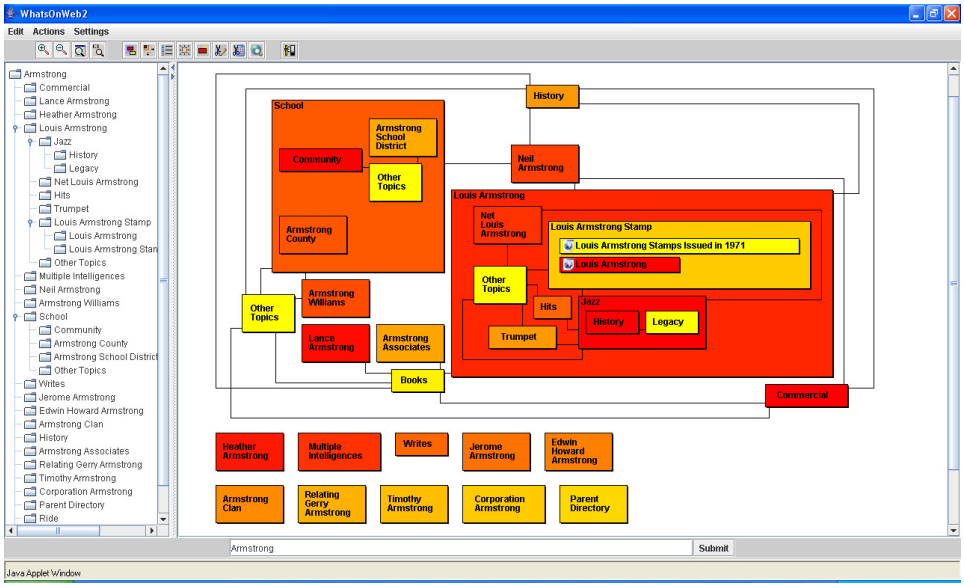
The drawing is computed using the orthogonal drawing algorithms of GDToolkit. The user can expand/contract clusters in the graph and the drawing changes accordingly. Using the constraint dimensions described in Section 18.5.3, each cluster is drawn as a box having the minimum size required to host just its label (if the cluster is contracted) or a drawing of its sub-clusters (if the cluster is expanded). The map in Figure 18.25 (a) shows a snapshot of the interface, where the results for the query "Armstrong" are presented; in the figure, the category "Louis Armstrong" has been expanded by the user. In order to preserve the user mental map during the browsing, `WhatsOnWeb` preserves the orthogonal shape of the drawing during after every expansion or contraction operation. For example, Figure 18.25 (b) shows the map obtained by expanding the categories "Jazz", "School", and "Louis Armstrong Stamp" in the first map.

18.6.3 Database Analysis

The third example of real-world application based on GDToolkit is focused on the analysis of a relational database. The *logical schema* of a relational database (also called *relational schema*) describes the database as a set of *tables*, where each table consists of a set of *attributes*. Links between tables might be present. A link between two tables A and B represents either an *integrity constraint* or a *join* between an attribute of A and an attribute of B; these two attributes are called the *attributes of the link*.



(a)



(b)

Figure 18.25 Snapshots of the user interface of `WhatsOnWeb`. (a) A map for the query "Armstrong"; in the map the user performed the expansion of the category "Louis Armstrong". (b) A subsequent map obtained by expanding the categories "Jazz", "School", and "Louis Armstrong Stamp"; this last category contains two URLs, described by reporting their titles. (Figure taken from [DL07].)

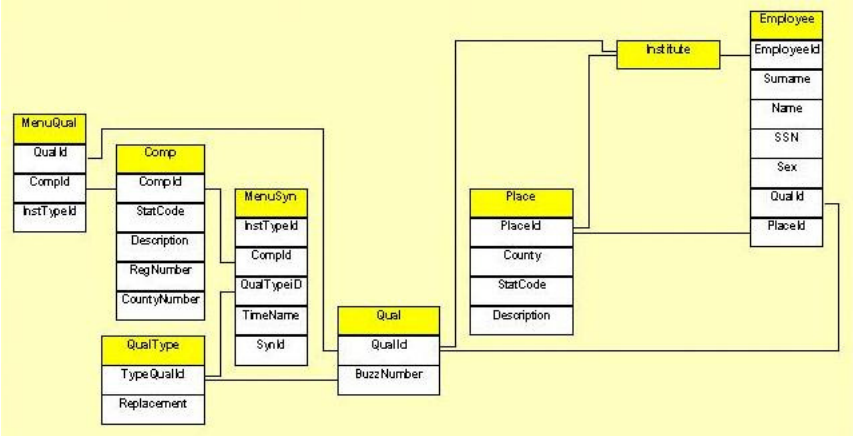


Figure 18.26 A database relational schema automatically drawn by `DBDraw`.

`DBDraw` [DDPP03, DDPP02] is a system that inspects a relational database and automatically computes a drawing of its relational schema (see, e.g., Figure 18.26). The drawing is represented within the orthogonal drawing convention subject to several constraints:

- Each table must be large enough to display inside it all its attributes.
- Each link connecting two tables A and B must be incident to A and on B in correspondence of the attributes of the link.
- Links cannot be incident to a table from north or from south.

The three constraints above are enforced by using the topology constraints and the metrics constraints described in Section 18.5.1 and Section 18.5.3.

Acknowledgements

Many people contributed to the development of GDToolkit, other than the authors of this chapter. We wish to warmly tank some of them whose contribution has been crucial for the success of the project. In alphabetic order, thanks to: Pier Francesco Cortese, Antonio Leonforte, Alessandro Marcandalli, Francesco Matera, Maurizio Patrignani, and Maurizio Pizzonia.

References

[AMO93] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, Englewood Cliffs, NJ, 1993.

[BBD+00] S.S. Bridgeman, G. Di Battista, W. Didimo, G. Liotta, R. Tamassia, and L. Vismara. Turn-regularity and optimal area drawings of orthogonal representations. *Computational Geometry: Theory and Applications*, 16:53–93, 2000.

[BBDL91] M. Beccaria, P. Bertolazzi, G. Di Battista, and G. Liotta. A tailorable and extensible automatic layout facility. In *Proc. IEEE Workshop on Visual Languages (VL '91)*, pages 68–73, 1991.

[BCPD04] J. M. Boyer, P. F. Cortese, M. Patrignani, and G. Di Battista. Stop minding your P's and Q's: Implementing a fast and simple dfs-based planarity testing and embedding algorithm. In *Proc. 11th Symposium on Graph Drawing, LNCS*, volume 2912, pages 25–36, 2004.

[BDD00] P. Bertolazzi, G. Di Battista, and W. Didimo. Computing orthogonal drawings with the minimum number of bends. *IEEE Trans. on Computers*, 49(8):826–840, 2000.

[BDD02] P. Bertolazzi, G. Di Battista, and W. Didimo. Quasi-upward planarity. *Algorithmica*, 32(3):474–506, 2002.

[BDLM94] P. Bertolazzi, G. Di Battista, G. Liotta, and C. Mannino. Upward drawings of triconnected digraphs. *Algorithmica*, 6:476–497, 1994.

[BDLN05] Carla Binucci, Walter Didimo, Giuseppe Liotta, and Maddalena Nonato. Orthogonal drawings of graphs with vertex and edge labels. *Comput. Geom.*, 32(2):71–114, 2005.

[CDD+02] A. Carmignani, G. Di Battista, W. Didimo, F. Matera, and M. Pizzonia. Visualization of the high level structure of the internet with HERMES. *Journal of Graph Algorithms and Applications*, 6(3):281–311, 2002.

[CGM+10] Markus Chimani, Carsten Gutwenger, Petra Mutzel, Miro Spönemann, and Hoi-Ming Wong. Crossing minimization and layouts of directed hypergraphs with port constraints. In *Graph Drawing*, volume 6502 of *Lecture Notes in Computer Science*, pages 141–152, 2010.

[DDGL05] E. Di Giacomo, W. Didimo, L. Grilli, and G. Liotta. A topology-driven approach to the design of web meta-search clustering engines. In *Theory and Practice of Computer Science (SOFSEM '05)*, volume 3381 of *Lecture Notes in Computer Science*, pages 106–116, 2005.

[DDGL06] E. Di Giacomo, W. Didimo, L. Grilli, and G. Liotta. Using graph drawing to search the web. In *13th International Symposium on Graph Drawing, GD 2005*, volume 3843 of *Lecture Notes in Computer Science*, pages 480–491, 2006.

[DDGL07] Emilio Di Giacomo, Walter Didimo, Luca Grilli, and Giuseppe Liotta. Graph visualization techniques for web clustering engines. *IEEE Trans. Vis. Comput. Graph.*, 13(2):294–304, 2007.

[DDLP10] Emilio Di Giacomo, Walter Didimo, Giuseppe Liotta, and Pietro Palladino. Visual analysis of one-to-many matched graphs. *J. Graph Algorithms Appl.*, 14(1):97–119, 2010.

[DDPP99] G. Di Battista, W. Didimo, M. Patrignani, and M. Pizzonia. Orthogonal and quasi-upward drawings with vertices of prescribed size. In *Symposium on Graph Drawing (GD'99)*, volume 1731 of *LNCS*, pages 297–310, 1999.

[DDPP02] G. Di Battista, W. Didimo, M. Patrignani, and M. Pizzonia. Drawing database schemas. *Software - Practice and Experience*, (32):1065–1098, 2002.

[DDPP03] G. Di Battista, W. Didimo, M. Patrignani, and M. Pizzonia. DBDraw - automatic layout of relational database schemas. In M. Jünger and P. Mutzel, editors, *Graph Drawing Software*, pages 237–256. Springer-Verlag, 2003.

[DETT99] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing*. Prentice Hall, Upper Saddle River, NJ, 1999.

[DGST90] G. Di Battista, A. Giammarco, G. Santucci, and R. Tamassia. The architecture of Diagram Server. In *Proc. IEEE Workshop on Visual Languages (VL '90)*, pages 60–65, 1990.

[Did00] W. Didimo. *Flow Techniques and Optimal Drawing of Graphs*. PhD thesis, Dipartimento di Informatica e Sistemistica, Univeristà di Roma"La Sapienza", 2000.

[Did05] W. Didimo. Computing upward planar drawings using switch-regularity heuristics. In *Theory and Practice of Computer Science (SOFSEM '05)*, volume 3381 of *LNCS*, pages 117–126, 2005.

[Did06] Walter Didimo. Upward planar drawings and switch-regularity heuristics. *J. Graph Algorithms Appl.*, 10(2):259–285, 2006.

[DK01] M. Dodge and R. Kitchin. *Atlas of Cyberspace*. Addison Wesley, 2001.

[DL98] G. Di Battista and G. Liotta. Upward planarity checking: "faces are more than polygons". In *Symposium on Graph Drawing (GD'98)*, volume 1547 of *LNCS*, pages 72–86, 1998.

[DL07] W. Didimo and G. Liotta. *Mining Graph Data*, chapter Graph Visualization and Data Mining, pages 35–64. Wiley, 2007.

[DP03] W. Didimo and M. Pizzonia. Upward embeddings and orientations of undirected planar graphs. *Journal of Graph Algorithms and Applications*, 7(2):221–241, 2003.

[DT96] G. Di Battista and R. Tamassia. On-line planarity testing. *SIAM Journal on Computing*, 25:956–997, 1996.

[EFK00] Markus Eiglsperger, Ulrich Fößmeier, and Michael Kaufmann. Orthogonal graph drawing with constraints. In *SODA*, pages 3–11, 2000.

[FCE95] Q. Feng, R. F. Choen, and P. Eades. How to draw a planar clustered graph. In *COCOON'95*, volume 959 of *LNCS*, pages 21–31, 1995.

[FG04] P. Ferragina and A. Gullí. The anatomy of a hierarchical clustering engine for web-page, news and book snippets. In *Fourth IEEE International Conference on Data Mining (ICDM'04)*, pages 395–398, 2004.

[FG05] P. Ferragina and A. Gullí. A personalized search engine based on web-snippet hierarchical clustering. In *14th international conference on World Wide Web*, pages 801–8106, 2005.

[FK96] U. Fößmeier and M. Kaufmann. Drawing high degree graphs with low bend numbers. In *Symposium on Graph Drawing (GD'95)*, volume 1027 of *LNCS*, pages 254–266, 1996.

[GKM08] Carsten Gutwenger, Karsten Klein, and Petra Mutzel. Planarity testing and optimal edge insertion with embedding constraints. *J. Graph Algorithms Appl.*, 12(1):73–95, 2008.

[MN95] K. Mehlhorn and S. Näher. LEDA: A platform for combinatorial and geometric computing. *Commun. ACM*, 38(1):96–102, 1995.

[MN00] K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing.* Cambridge University Press, Cambridge, UK, 2000.

[Tam87] R. Tamassia. On embedding a graph in the grid with the minimum number of bends. *SIAM J. Comput.*, 16(3):421–444, 1987.

[Tam98] Roberto Tamassia. Constraints in graph drawing algorithms. *Constraints*, 3(1):87–120, 1998.