Journal of Graph Algorithms and Applications
http://jgaa.info/ vol. 17, no. 4, pp. 495–513 (2013)
DOI: 10.7155/jgaa.00301

Graph Drawing in Ti*k*Z

Till Tantau

Institute of Theoretical Computer Science
Universität zu Lübeck
D-23562 Lübeck, Germany

Abstract

At the heart of every good graph drawing algorithm lies an efficient procedure for assigning canvas positions to a graph's nodes. However, any real-world implementation of such an algorithm must address numerous problems that have little to do with the actual algorithm, like handling input and output formats, formatting node labels, or styling nodes and edges. We present a new framework, written in the Lua programming language, that allows implementers to focus on core algorithmic ideas and leave all other aspects to the framework. Algorithms implemented for the framework can be used directly inside the Ti*k*Z graphics language and profit from the capabilities and quality of the TEX typesetting engine. The framework comes with implementations of standard tree drawing algorithms, a modular version of Sugiyama's layered algorithm, and several force-based multilevel algorithms.

Submitted:	Reviewed:	Revised:	Accepted:	Final:
December 2012	February 2013	March 2013	March 2013	April 2013
		Published:		
		July 2013		
	Article type:		Communicated by:	
	Regular paper		W. Didimo and M. Patrignani	

This is an extended version of the paper [15] presented at the Graph Drawing 2012 conference.
E-mail address: tantau@tcs.uni-luebeck.de (Till Tantau)

1 Introduction

A graph drawing algorithm is, at its core, a way of mapping graphs to drawings of graphs. The idea underlying an algorithm can be very simple, but it is a long way from just an idea to a complete system for drawing graphs that allows the configuration of node distances, preferred edge slopes, or the font used for text labels in nodes. This "long way" consists of three main steps: First, the input graphs need to be *specified* in some way. Typically, a syntax is defined that authors[1] must use to describe the graphs they wish to be drawn, and the system must be able to parse the syntax to construct internal representations of input graphs. Second, the algorithm itself needs to be *implemented* in some programming language. Third, the computed drawing of the graph must be *rendered* in such a way that authors can further process the result. Typically, the output is a vectorized or bitmap drawing in some standard format like PDF or PNG that can then be used for printing or inclusion in a document.

A major obstacle to implementing new graph drawing algorithms is that researchers are typically forced to address all three of the above problems, even though they would like to focus on the implementation part. This leads to interesting graph drawing algorithms being available only as prototypes that lack many features necessary to make them usable in practice (an example is the force-based Lombardi graph drawer presented at the Graph Drawing 2011 conference [2]). Even when algorithms are part of powerful toolkits, these toolkits may be difficult to integrate into a typesetting workflow (as is the case for MATHEMATICA or the Open Graph Drawing Framework [3]) or extending them by new algorithms may be a nontrivial software engineering problem (as for the GRAPHVIZ toolkit [7]). Keeping even simple styling parameters like font sizes or arrow tips consistent across several drawings is a major problem when different systems are used (and often even when the same system is used).

The present paper introduces a new framework for implementing graph drawing algorithms that aims at letting implementers focus on the implementation of new algorithmic ideas by handling the other steps. In particular, implemented algorithms can immediately be used by authors within a widely used typesetting system, namely TeX. The framework augments an existing graphics description language, called "Ti*k*Z"[2] [14], by graph drawing facilities. This language allows authors to specify graphics directly inside TeX documents using special macros and the graphics are produced on-the-fly during a run of the TeX program on the manuscript. As an example, to draw a graph using the Sugiyama method [6, 13], authors add the option "`layered layout`" to the description of the graph as demonstrated in Figure 1.

While the framework makes it easy for authors to apply powerful graph drawing algorithms to graphs specified inside a TeX document, its main pur-

[1]Users who employ a graph drawing framework to draw graphs will be referred to as *authors* in the following, while users who implement algorithms on top of a graph drawing framework will be referred to as *implementers.*

[2]Ti*k*Z means "Ti*k*Z ist *kein* Zeichenprogramm," a recursive German acronym in the tradition of "GNU is Not Unix" cautioning that Ti*k*Z is *not* a graphical editor.

```
% Somewhere in the TeX manuscript
Consider the diagram

\tikz \graph [layered layout] {
  a -> {b, c, d} -- e <-> a
};
```

Consider the diagram

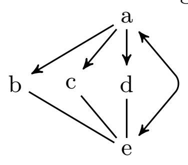


Figure 1: On the left a typical excerpt from a manuscript is shown; running TEX on it results in the output shown on the right.

pose is to allow the rapid implementation of new graph drawing algorithms by researchers from the field. Such algorithms must be implemented in Lua [9], a light-weight, well-designed scripting language in which the complete framework is written. Lua was chosen, firstly, because it is part of modern versions of TEX and both the framework and new algorithms work out-of-the-box on all systems running TEX. Secondly, the speed of Lua is perfectly sufficient for drawing small to medium-size graphs on-the-fly during a run of the TEX program, see the conclusion for some benchmark results. Thirdly, libraries written in C can both be accessed and loaded dynamically by Lua, making C code a viable, but less portable option for time-critical parts of graph drawing algorithms for large graphs and for integrating algorithms already implemented in C or C++.

The framework treats graph drawing algorithms as transformations from one class of graphs to another, a design principle advocated by Di Battista et al. [5]. Graph drawing algorithms declare which kinds of graphs they accept as input and which kind of graphs they produce as output, and the framework will automatically apply appropriate pre- and post-transformations to ensure that any input graph can be given as input and be used with any algorithm.

Even though the seamless integration of the framework with Ti*k*Z was an important design goal, the framework's architecture treats TEX as just one possible system from which it receives to-be-drawn graphs and to which it sends back canvas positions for the nodes. The framework can also be linked to systems such as interactive graph editors or command line tools.

Related Work. One of the earliest systems that integrates a graph drawing algorithm into TEX dates back to 1989: the TreeTEX program of Brüggemann-Klein and Wood [1] is an improved version of the Reingold–Tilford algorithm implemented directly in the TEX programming language. Since then, dozens of both in- and interdependent packages have been developed in the context of TEX that implement a variety of specialized graph drawing algorithms covering different fields. For instance, there are dedicated packages for drawing pedigree trees in medicine [16] or message sequence charts in software engineering [10]. The packages all come with their own special-purpose syntax and design philosophies and they differ strongly with respect to their portability.

The system presented in the following seems to be the first serious attempt at augmenting a general-purpose graphics description language like POSTSCRIPT, PSTRICKS, METAFONT, or SVG by general graph drawing facilities. One could argue that the MATHEMATICA system is an example of treating graph drawing as a subtask of producing documents (called "notebooks" in MATHEMATICA), but describing graphics and typesetting documents do not lie at the heart of MATHEMATICA.

Concerning the range of algorithms implemented, the framework is roughly comparable to the GRAPHVIZ toolkit [7], except for the algorithms for radial graph drawings, which are still missing. Compared to the Open Graph Drawing Framework (OGDF) [3], many of OGDF's advanced support algorithms (like computing SPQR decompositions) are not available. In both cases, the fact that GRAPHVIZ and OGDF use compiled code makes them much faster than our Lua implementation.

Organisation of this Paper. This paper describes graph drawing in Ti*k*Z from three different perspectives: the author's perspective, the integration perspective, and the implementer's perspective.

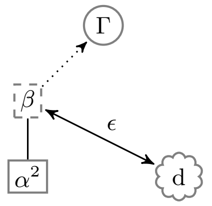
Following a brief review of the graphics description language Ti*k*Z below, we explore the syntax authors can use in Ti*k*Z for the description of graphs; see Section 2. I will argue that choices in the syntax are not only a matter of taste, but can influence the quality of graph drawings. In Section 3 on integration issues, we contrast the extension of an existing graphics description language by graph drawing capabilities to the approaches taken by other graph drawing systems. Section 4 is addressed at implementers of graph drawing algorithms, explaining the framework's design and architecture. This section contains a complete, commented implementation of a simple graph drawing algorithm. We will see how the framework can be asked to perform a number of automatic graph transformations that allow the implementer to focus on the core algorithmic idea and write lean code.

The Graphics Description Language Ti*k*Z in a Nutshell. Ti*k*Z is a vector graphics description language implemented as a TeX macro package. It has been under continuous development for the last ten years and comes with over one thousand pages of documentation. Being part of any standard TeX installation and being written entirely as a collection of macros, it can be used out-of-the-box on any system running the TeX program.

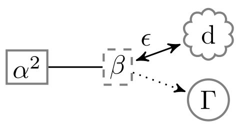
Ti*k*Z' syntax borrows from METAFONT, a graphic description language designed by Donald Knuth, and PSTRICKS, a macro package similar to Ti*k*Z but tailored specifically to the POSTSCRIPT language, which is widely used in printing industry. The macros of Ti*k*Z convert the description of a graphic into a stream of low-level primitives for the output format of the specific version of TeX used. In particular, PDF, POSTSCRIPT, and even SVG output are directly supported.

The most important feature of Ti*k*Z for the purposes of the present paper will be its support for specifying *nodes* and *edges* between them. Nodes can have one of many possible shapes (the libraries define dozens of possible shapes, and arbitrarily complex new shapes can be defined) and edges can be routed in different manners. Nodes have *anchors*, which are coordinates inside the node that can be referenced later and which are similar to the *ports* of nodes common in the context of graph drawing. The below code shows a typical, but admittedly artificial example of creating and connecting nodes using the Ti*k*Z syntax.

```
\tikz {
  \node (a)          at (0,0) {$\alpha^2$};
  \node (b) [dashed] at (0,1) {$\beta$};
  \node (c) [circle] at (1,2) {$\Gamma$};
  \node (d) [cloud]  at (2,0) {d};
  \draw (a) edge                        (b);
  \draw (b) edge [->,  dotted]          (c);
  \draw (b) edge [<->, "$\epsilon$"]  (d); };
```



Above, coordinates for the nodes have been specified explicitly in centimeters. The main purpose of the graph drawing framework described in the following is to compute these positions for us. Leaving out the `at`-parts and adding the options `[tree layout, grow=right]` after `\tikz` on the first line yields the result shown on the right.



2 The Author's Perspective: Specifying Graphs and the Quality of Graph Drawings

Graph drawing systems should make it easy for authors to specify the graphs that they wish to be drawn. Examples of graph description languages include GRAPHML, an XML-based markup language; the DOT format used by GRAPHVIZ; or the GML format used by the Open Graph Drawing Framework. Graphs can also be specified indirectly as the results of computations, as in computer algebra systems like MATHEMATICA, allowing succinct graph specifications of large graphs.

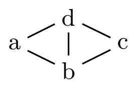
Desirable Properties of Graph Description Languages in the Context of Graph Drawing. It may seem to be largely a matter of taste which format is used for specifying graphs; graph drawing algorithms, including those implemented using the framework presented in this paper, internally work on an abstract representation of the graph anyway, namely on a set of nodes and a set of edges. However, especially for graphs specified by humans as part of a manuscript, there are several reasons why a format needs to be chosen carefully.

Firstly, authors should be able to provide hints to graph drawing algorithms by means of syntax. An important hint is actually the order in which nodes and edges are specified by an author; it typically has semantic meaning. The importance of this information is well-recognized; for instance, experiments by Gansner et al. [8] have shown that cycle removal, a first step in algorithms for drawing layered graphs, should be based on a depth-first traversal of the input graph as specified by the author rather than on using, say, randomized methods (see for instance [5]) that do not take the graph description into account. The quality of cycle removals can be further improved if authors can indicate "backward edges" in a natural way. While the node ordering is implicit in almost any format, specifying hierarchical dependencies or special edge kinds can be cumbersome or even impossible. To illustrate the subtleties involved, consider the problem of using a general-purpose graph description language to specify ternary trees in which nodes may have "missing" children, like a missing left or middle child, for which space needs to be reserved in the layout.

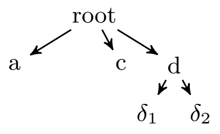
Secondly, while the above considerations aim at improving the quality of the results produced by graph drawing algorithms, a good format will also make it easy for authors to specify a graph in a succinct and self-explaining manner. Authors will prefer to write (and, later on, also to read) `a -> b -> c -> a`, as in the DOT format, over having to first create three nodes (using `\node` thrice in TikZ or the `<node>` tag in GRAPHML or `node[...]` in GML) followed by having to specify three edges (using `edge`, `<edge>`, or `edge[...]`). Styling options, label texts, and "hints" to graph drawing algorithms should also be easy to specify.

A Graph Description Language for Graph Drawing in TikZ. All of the existing formats have drawbacks: The standard way of specifying nodes and edges in TikZ using the `\node` command is too verbose and the same is true for GRAPHML and all other XML-based formats. The DOT format used in the GRAPHVIZ toolkit is more concise and defines a number of ways of styling nodes and edges, but this set of options is neither extensible nor extensive. Moreover, styling options cannot always be provided next to the object to which they apply. As a consequence, a new format was developed that is tailored to the specific needs of graph drawing in TikZ (but the standard syntax and other formats can also be used). The basic syntax of this new format leans on the DOT format and graphs specified using only the basic features of DOT can be processed directly. Nodes can be grouped and connected hierarchically as in the following examples:

```
\tikz [spring layout, vertical=d to b]
  \graph {a -- b -- c -- d -- a; b -- d};
```

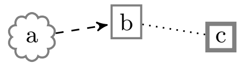


```
\tikz [tree layout] \graph {
  root -> { % second child is "missing"
    a, , c, d -> { "$\delta_1$", "$\delta_2$" }
}};
```



The first main difference to the DOT format is the place where options are specified: The `\graph` command, each node, and also each edge indicator (strings like `--` or `->`) can be followed by options written in square brackets. In particular, in a sequence of nodes connected by edges, each node and each edge can have its own options without any need for repeating the node names:

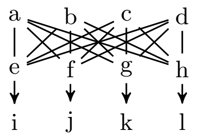
```
\tikz [tree layout, grow=right]
  \graph { a[cloud] ->[dashed]  b[nudge up=2mm]
                    --[dotted]  c[ultra thick] };
```



By attaching appropriate options to nodes, we can easily fine-tune graph drawings: In the example, `nudge up` raises ("nudges up") `b`'s placement by two millimeters relative to the position computed by the algorithm.

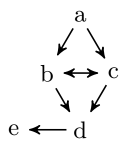
The second main difference concerns the semantics of edge indicators, of which there are five (`--`, `->`, `<-`, `<->`, and the special `-!-` meaning "remove an edge specified earlier"). While graph descriptions in DOT format are sequences of edges plus syntactic sugar (we can write `a -> b -> c` instead of `a -> b; b -> c` and `a -> {b; c}` instead of `a -> b; a -> c`), we now interpret graph descriptions as *graph expressions*, that is, as terms whose atoms are single vertices and whose function symbols combine subgraphs to larger graphs. The text `{a, b, c} -- {d, e, f}` is interpreted as the term $\gamma(G_1, G_2)$ where $G_1 = (\{a, b, c\}, \emptyset)$ and $G_2 = (\{d, e, f\}, \emptyset)$ are two discrete three-node graphs and γ is a *combining function* that takes the union of the two (possibly overlapping) graphs and adds some edges. Which combining function is used can be changed for each use of an edge indicator, allowing authors to describe graphs in a succinct and, ideally, self-explaining manner. In the following example, the first two layers form a complete bipartite graph as prescribed by the first combining function, while the last two layers form a matching.

```
\tikz \graph [layered layout] {
  { a, b, c, d } --[complete bipartite]
  { e, f, g, h } ->[matching]
  { i, j, k, l }
};
```



Options attached to nodes or edges provide local information to graph drawing algorithms. In order to communicate information about larger structures inside the input graph, authors can specify *subgraphs* of the input graph. For instance, hyperedges can be modeled as discrete subgraphs, just as the "same rank clusters" of the DOT format. "Clusters of edges" can also be regarded as subgraphs whose edges should be routed similarly by an edge routing algorithm. To handle all of these and future applications in a uniform manner, implementers can declare *subgraph kinds* like "`hyper`" for hyperedges or "`same layer`" for same layer clusters. Using an option key like `same layer` at the beginning of a group indicates that everything inside the group is part of a subgraph.

```
\tikz \graph [layered layout] {
  a -> { b, c } -> d -> e;
  {[same layer] b <-> c };
  {[same layer] d, e };
};
```



Authors can specify that on certain subgraphs a graph drawing algorithm differing from the main algorithm should be used. Running the different algorithms and resolving conflicts are handled automatically by the framework.

Making Syntactic Structure Visible to Graph Drawing Algorithms. Just like any other general-purpose graph drawing framework, our framework internally works on abstract representations of the input graphs that do, however, include all of the syntactic "hints" given by the author in a format-independent way. For instance, we store the options trailing a node or an edge as tables attached to the node's or edge's representing object. We use arrays to store nodes and edges in the order in which they appear. Information about nodes that are "not there" is communicated by a string of *events* that are generated by the parser whenever "something interesting happens" (such as "missing node encountered"). For each of the above-mentioned subgraph kinds, a separate array stores all encountered subgraphs of this type.

3 The Integration Perspective: Graph Drawing and Typesetting

Typesetting a document and drawing a graph are generally treated as two unrelated problems. On the one hand, we have specialized software and formats for typesetting documents, like Donald Knuth's TEX, Adobe's INDESIGN, or Apache's OPENOFFICE; on the other hand, we have specialized software and formats for drawing graphs, like GRAPHVIZ, the graph drawing packages inside MATHEMATICA, or the Open Graph Drawing Framework. This separation into two distinct spheres of development allows for very efficient and specialized implementations. The separation, however, also implies the need to establish an interface between the two worlds.

Interfacing Between a Graph Drawer and a Typesetter. The typical interface between a graph drawer and a typesetter is simple and one-way: At some point, a document author will run the graph drawing program (manually or triggered by a "makefile"), resulting in drawings in a vectorized format like PDF or SVG or in a bitmap format like PNG. These drawings must then somehow be embedded into the document. Unfortunately, it is hard for authors to ensure that the drawings produced by the graph drawing system match the style sheet used for the main document. Basic requirements like matching colors can often be met, but already font sizes are harder to get right. Meeting advanced requirements such as including mathematical text in node labels is typically impossible even when powerful graph drawing systems are used; let alone when special-purpose, standalone graph drawing systems are employed that are tailored for one specific visualization task.

Many of these problems can be avoided by using the graph drawing system only for computing positions for the nodes of a graph and leaving the rendering of the graph to the typesetting system; for instance using METAFONT or PSTRICKS (or Ti*k*Z for that matter) in conjunction with TeX. Unfortunately, this approach entails a new problem: The graph drawing system now misses information concerning the sizes of text labels, which is vital for many graph drawing algorithms.

Integrating a Graph Drawer into a Typesetter. The graph drawing framework for Ti*k*Z sidesteps the indicated problems by integrating the typesetter with the graph drawer. No separate program is used for the graph drawing; rather, the framework is called directly from the typesetting program TeX. The framework is written in Lua, a language that has been integrated into current versions of TeX. It is this recent addition to the TeX program that makes graph drawing in Ti*k*Z feasible: Although TeX is Turing-complete, programming directly in TeX is an arcane art practised by only a few devout disciples. In contrast, Lua is a small, elegant, and fast scripting language that is easy to use both for beginners and experts.

To get a feeling for how the integration works, let us consider the command `\tikz \graph[tree layout] {a -> b -> {c, d}};`. Upon encountering the special option `tree layout`, Ti*k*Z tells the framework that a "graph drawing scope" has started. This "telling" is done by calling a method of the framework's class `InterfaceToDisplay`. The class encapsulates all of the functionality offered by the framework to a system like TeX, whose main job from the framework's point of view is to "display" graphs – hence the class' name.

The parsing of the graph inside the braces results in four nodes and three edges being created. Each of the four nodes is rendered normally, resulting in TeX boxes that contain all of the low-level primitives needed to render the nodes, including label texts, borders, shadows, and whatever else might have been specified. Normally, such a box would now be added to the page; but inside a graph drawing scope, the box gets intercepted at this point and passed down to the framework (again, by calling methods of the interface class). The contents of the box are stored in an internal table, together with detailed size information. From TeX's point of view, the box disappears at this point. In contrast, edges are not rendered when they are encountered. Instead, only the information where an edge starts, where it ends, and the edge's local options are passed down.

At the end of the graph drawing scope, a complete description of the graph, consisting of all nodes, including their exact convex hulls, all edges, including their labels and options, will have accumulated inside the framework in the form of Lua tables. At this point, the framework switches over completely to Lua

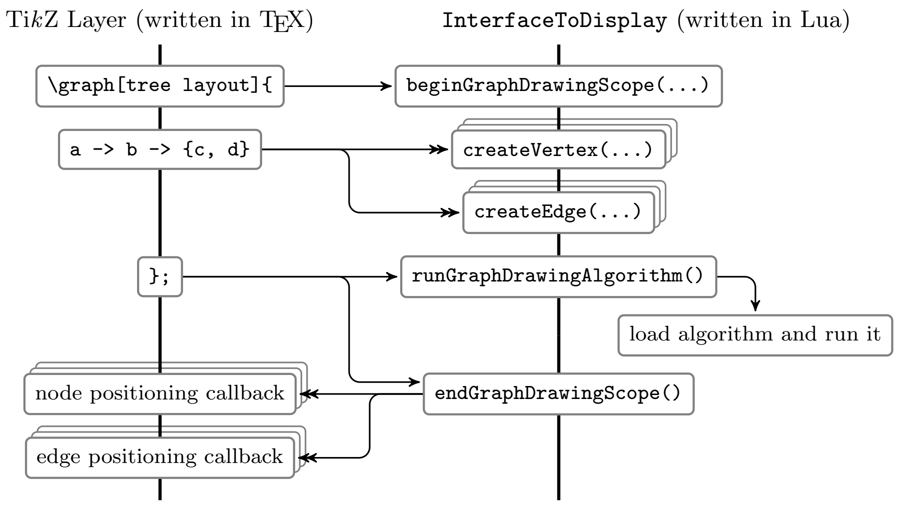

TikZ Layer (written in TEX)
InterfaceToDisplay (written in Lua)
\graph[tree layout]{
beginGraphDrawingScope(...)
a -> b -> {c, d}
createVertex(...)
createEdge(...)
};
runGraphDrawingAlgorithm()
load algorithm and run it
node positioning callback
endGraphDrawingScope()
edge positioning callback

Figure 2: Call graph of the interaction between TikZ and the interface of the framework. Naturally, this figure has been created using graph drawing in TikZ: The Sugiyama method was applied to the call graph and `same layer` was used with all node pairs that needed to be aligned horizontally. For the edges between nodes on different layers, TikZ's capability of rendering edges using only horizontal and vertical segments was harnessed.

and runs the graph drawing algorithm corresponding to the option `tree layout` (this happens to be the Reingold–Tilford algorithm). When the algorithm terminates, the node and edge objects are storing the computed canvas coordinates. Since, from the typesetter's perspective, the nodes and edges "disappeared" during the parsing of the graph drawing scope, they need to be reinserted into the page and, only now, the edges, too, get rendered. The call graph in Figure 2 visualizes this back-and-forth between the typesetter and the framework.

Although nodes disappear and reappear from the typesetter's memory, from TEX's perspective there is no difference between nodes inside graph drawing scopes and nodes positioned without the use of the graph drawer. In particular, the nodes inside a graph drawing scope can be referenced from outside this scope just like any other node, allowing the seamless integration of drawn graphs into larger graphics, see Figure 3.

4 The Implementer's Perspective: Lean Implementations of Graph Drawing Algorithms

Whether a new idea of how to draw graphs really works in practice can only be tested by implementing a prototype. Such prototypes then need to be expanded or even rewritten if authors are really supposed to use them. In the following, we

```
\tikz {
  \graph [spring electrical layout,
          nodes={circle, ball color=white},
          edges={decoration=coil, decorate} ] {
    1 -- 2 -- {3, 4, 5};  4 -- 3 -- 5 -- 6 -- 3;
  };
  \scoped[on background layer]
    \filldraw [black!20, line width=2.5em,
               line join=round]
      (2.center) -- (4.center) -- (3.center) --
      (5.center) -- cycle; }
```

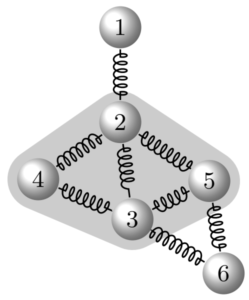


Figure 3: Example of a picture where the graph drawing system is used to first render a six node graph using a spring electrical layout. Subsequently, the positioned nodes can be referenced normally such as in the code that adds the shaded area on the background.

have a look at how implementers are helped by the graph drawing framework with turning prototypes into implementations that create publication-quality graph drawings.

Graph Drawing as a Sequence of Transformations. The design of the graph drawing framework adheres to the philosophy advocated by Di Battista et al. [5] of seeing graph drawing as a series of graph transformations. We start with an "arbitrary" input graph and end with a graph whose nodes and edges are embedded in the plane. For instance, the popular Sugiyama method for drawing layered graphs consists of first decomposing the input graph into connected components, then transforming each component into a directed acyclic graph, followed by a whole series of further transformations. In contrast, most force-based algorithms will also decompose the input graph but will then turn each component into a simple, undirected graph. Other typical transformations include planarization or the decomposition of the graph into singly, bi-, or tri-connected components.

The principle of treating graph drawing as graph transformations is reflected by the way graph drawing algorithms must be implemented. Each new algorithm is actually just a transformation and it must declare which kinds of graphs the algorithm expects and what kind of graphs it will output, using a Lua table stating preconditions the graph must meet and postconditions the graph will meet. For instance, the precondition "works only on connected graphs" tells the framework that it must first decompose the input graph into connected components and that these components must be passed to the algorithm individually. Similarly, the precondition "needs a spanning tree" tells us that the algorithm will work on trees (or at least will need a spanning tree in addition to the original graph). Other properties concern the output graphs rather than the input: The postcondition "upward oriented" tells the framework that the result of the transformation is a layered drawing where nodes on the same layer have the same y-coordinate and the larger the layer index, the larger the y-coordinate.

Pre- and postconditions give algorithm designers fine control over which transformations are applied to a graph before and after the actual algorithm runs. Standard transformations include decomposition into connected components, shifting and rotating to meet author-specified anchoring and orientation requirements, as well as rotating tree-like graphs so that they "grow" at an author-specified angle. The last transformation is especially useful since graph drawing algorithms for trees and layered graphs only need to handle the case that a tree grows upwards and the framework will automatically rotate the graph when a user requests that the tree should grow in another direction. Having access to the convex hulls of the nodes, the framework can even correctly compensate for the effects of rotating a drawing of a graph, but not rotating the individual nodes.

Framework Architecture: Separation of Concerns. An implementation of a graph drawing algorithm should neither have to worry about how graphs are specified nor about how transformations are applied to them prior or after the algorithm has run. Rather than just encouraging algorithms to be independent of the internal workings of the framework, its layered architecture actually enforces it:

1. The *display layer* interacts with the *display system* that the author uses. In the framework's terminology, a display system is any software that "displays graphs" such as TEX in conjunction with Ti*k*Z; but a graphical editor would also constitute a display system.

   All functionality offered by the framework to a display system is encapsulated in the class `InterfaceToDisplay` mentioned earlier. While this class includes the bulk of the code necessary for the interaction with an arbitrary display system, each display system requires a small amount of specific "binding" code in a special "binding class." The framework was carefully designed so that nothing TEX-specific is used or even just assumed anywhere except inside the class binding to TEX; in particular, the framework can be used completely independently of TEX, provided an appropriate binding is created.

2. Implementers of graph drawing algorithms see the *algorithm layer* of the graph drawing framework. It consists of Lua classes for graphs, nodes, and edges, as well as libraries for common algorithmic tasks such as graph traversal, graph decomposition, and so on.

   Graph drawing algorithms can neither access nor influence the communication between the framework and the display layer. Rather, they must use the class `InterfaceToAlgorithms`, which is a counterpart to the interface of the framework to display systems. The fact that algorithms cannot access the display system makes sure that algorithms implemented for the framework work with all display systems.

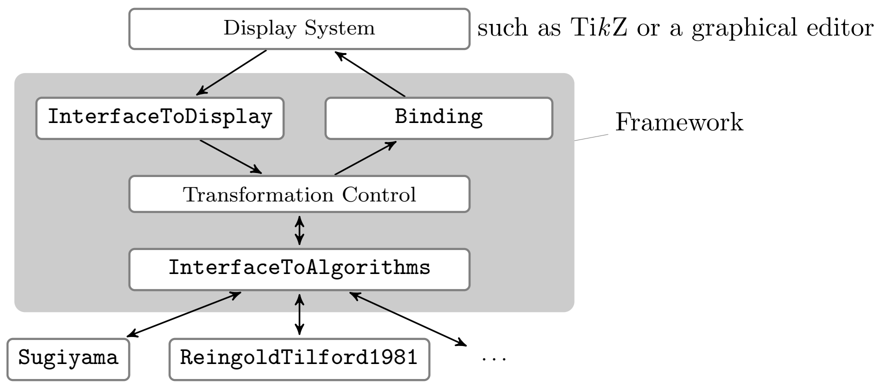


Figure 4: Overview of the framework architecture. Arrows indicate function calls. The framework is made up by the shaded area; the display system at the top as well as the example algorithms at the bottom *use* the framework, but they are not part of it. Like Figure 2, this figure has been created using Ti*k*Z in conjunction with the layered graph drawing layout. Observe how the overall styling of Figure 2 is consistent with the above figure, even though they depict quite different kinds of graphs.

An important functionality offered by the `InterfaceToAlgorithms` class is the *declaration* of algorithms and options together with their properties and even documentation at runtime. Display systems can use this information to display lists of available algorithms and options together with their documentation.

3. The heart of the framework is the *transformation control;* the other layers communicate with this control. Its job is to take the graph description as communicated by the display system through the `InterfaceToDisplay` and apply the necessary series of graph transformations to it. The most important transformation will be the call of the algorithm(s) selected by the author, but other transformations will also be performed as indicated earlier. Once the input graph has been transformed into a graph embedded in the plane, the computed positions are passed back to the display system through its binding.

The main aspects of the architecture of the framework are depicted in Figure 4.

Example Implementation of a Tree Drawing Algorithm. The graph drawing framework aims at facilitating lean graph drawing algorithm implementations that focus on the core ideas. What this means in practice will be demonstrated in the following by presenting an example implementation of a tree drawing algorithms.

The “core idea” behind this algorithm is to recursively apply the following rule to each node: Consider the bounding boxes of the trees rooted at the node’s children. Arrange these boxes from left to right with a fixed padding between them. Then center the node below these drawn trees. (This placement strategy is not particularly clever, unforgivingly ignores the node sizes, and only used for the purposes of this example.)

The algorithm’s code, which we typically place in its own file, starts with the creation of an (empty) algorithm class and the declaration of its properties. For the declaration, we first need to import the key interface function `declare`.

```
-- File TreeExample.lua
local declare = require "pgf.gd.interface.InterfaceToAlgorithms".declare

local TreeExample = {}  -- As yet empty algorithm class,
                        -- methods are defined later
declare {
  key            = "tree example layout", -- name for the display layer
  algorithm      = TreeExample, -- bind the name to this class
  preconditions  = { tree = true },
  postconditions = { upward_oriented = true },
  summary        = "A tree layout algorithm for demonstration purposes"
}
```

The precondition in the declaration ensures that when the framework invokes the `run` method, the algorithm only gets a tree as input – even if the original input graph is neither acyclic nor even connected: the algorithm will be invoked once for each connected component with a spanning tree of the component as input. As with all other transformations, the to-be-used algorithm for computing spanning trees can be selected by authors through options.

The `run` method just calls a recursive placement function, which we will define in a moment and whose parameters are the to-be-drawn tree (`self.spanning_tree`), the root node of this tree, a horizontal position for the tree’s left-most node, and a vertical position for the root:

```
local recursion -- This is a forward declaration
function TreeExample:run() -- This method gets called by the framework
  recursion(self.spanning_tree, self.spanning_tree.root, 0, 0)
end
```

The recursive method first reads the desired distance of the root node to its children from the node’s option table. Such tables are created during the parsing and get attached to all vertices, edges, and to the graph as a whole; the framework provides default values when the author has not specified an option’s value explicitly.

```
function recursion(tree, root, left_end, y)
  local level_dist     = root.options['level distance']
  local sibling_dist   = root.options['sibling distance']
```

Now comes the implementation of the core idea: The algorithms loops over all outgoing edges of the root and recursively draws the trees rooted at the "heads" of these edges (the children of the root). Each recursive call returns the position of the rightmost position used, which we increase by the sibling distance and use as the left end of the next child tree.

```
  local outgoing_edges  = tree:outgoing(root)
  local right_end       = left_end
  for i,edge in ipairs(outgoing_edges) do -- a Lua for-loop
    right_end = recursion(tree, edge.head, right_end, y + level_dist)
    if i < #outgoing_edges then -- pad all but last node
      right_end = right_end + sibling_dist
    end
  end
```

The last step is to center the root by modifying the root's position field:

```
  root.pos.x = (left_end + right_end) / 2
  root.pos.y = y
  return right_end
end
```

We can immediately use the code as demonstrated in Figure 5.

```
\usegdlibrary{TreeExample} % Loads the Lua file

\tikz [tree example layout,
       grow'=45, sibling distance=2em]
  \graph {
    a -> {b, c -> {d, e}, f -> g},
    b -- d
  };
```

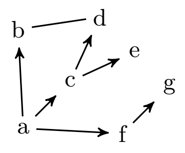


Figure 5: Example use of the example tree drawing algorithm in a TEX document. The algorithm requires its input to be a tree, which is not the case here, and draws this tree "upwards," which is not desired by the author. The framework automatically removes the `b--d` edge to turn the input graph into a tree and applies transformations to honour the requested 45° direction of "tree growth."

Naturally, the implementations of the "real" graph drawing algorithms that are already part of the framework are more complex than the above example: The implementation of the Reingold–Tilford [12] tree drawing method has 130 lines of code; the modular implementation of the Sugiyama method [6, 13] needs a bit over 2000 lines of code, nearly half of which implement the network simplex algorithm [4]; implementations of different force-based algorithms need between 350 and 500 lines of code and another 500 lines for supporting multilevel approaches.

5 Conclusion

The framework presented in this paper allows researchers to rapidly implement and test new graph drawing algorithms, which authors can immediately use to produce high quality drawings of graphs that are part of a larger text. Since the framework is fully integrated into a graphics description language, the full power of the language can be used to modify and augment graph drawings. Being part of the main Ti*k*Z code trunk, the framework is available through `sourceforge.net/projects/pgf` and will automatically be part of future standard TEX distributions.

Since the core of the graph drawing framework is implemented in Lua, it is much faster than equivalent code written in TEX, but much slower than code written in compiled languages such as C. For smaller graphs, Lua implementations of graph drawing algorithms turn out to be sufficiently quick: TEX needs 3.2 seconds on a 2.8GHz CPU to typeset the present paper, but only 0.5 of these 3.2 seconds are needed by the Lua graph drawing algorithms to draw the eleven graphs of the present paper. A larger example, the 47-node graph from the GRAPHVIZ example suite that depicts the Unix history, already needs 1.2 seconds to draw and even larger graphs drawn using force-based methods may need minutes or hours to process. This problem can partly be sidestepped by using Ti*k*Z's built-in facilities for only (re)processing pictures whose syntactic description has changed in the manuscript. This drops the total processing time for the present manuscript to 1.2 seconds. Alternatively, time-critical parts of algorithms can be implemented in C libraries that get loaded dynamically by Lua; but this entails the portability and deployment problems that come along with compiled C code.

Concerning the future of the framework, apart from adding more standard or experimental algorithms, three avenues of development seem particularly promising: First, the integration of the Open Graph Drawing Framework would immediately make a large number of sophisticated (and fast) algorithms available to authors. A first implementation in the source repository demonstrates the great potential of this integration, but the deployment of the compiled C code is, indeed, tricky. Second, a command-line interface for the framework would make integration with non-TEX-based workflows easier. From the framework's point of view, the command-line interface would constitute a display system and an appropriate binding would have to be implemented. Finally, I would like to recommend more research on graph drawing algorithms that address the drawing of small graphs specified by humans in a graph description language.

Acknowledgements

The first prototype of the framework was implemented as a graduate project in 2010 by Renée Ahrens, Olof-Joachim Frahm, Jens Kluttig, Matthias Schulz, and Stephan Schuster. The prototype was greatly extended by Jannis Pohlmann while working on his Diploma thesis [11], in which he details his implementations of the modular Sugiyama algorithm and of the multilevel force-based methods.

I would also like to very much thank the anonymous reviewers for their valuable feedback and careful proofreading of this paper.

References

[1] A. Brüggemann-Klein and D. Wood. Drawing trees nicely with TEX. *Electronic Publishing*, 2(2):101–115, 1989.

[2] R. Chernobelskiy, K. I. Cunningham, M. T. Goodrich, S. G. Kobourov, and L. Trott. Force-directed Lombardi-style graph drawing. In *Proceedings of the 19th International Conference on Graph Drawing, (GD 2011)*, pages 320–331, Berlin, Heidelberg, 2012. Springer-Verlag. `doi:10.1007/978-3-642-25878-7_31`.

[3] M. Chimani, C. Gutwenger, M. Jünger, K. Klein, P. Mutzel, and M. Schulz. The Open Graph Drawing Framework. Poster at the 15th International Symposium on Graph Drawing 2007 (GD 2007), 2007.

[4] W. H. Cunningham. A network simplex method. *Mathematical Programming*, 11(1):105–116, 1976. `doi:10.1007/BF01580379`.

[5] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing, Algorithms for the Visualization of Graphs*. Prentice Hall, 1999.

[6] P. Eades and K. Sugiyama. How to draw a directed graph. *Journal of Information Processing*, 13(4):424–436, 1990.

[7] J. Ellson, E. Gansner, E. Koutsofios, S. North, and G. Woodhull. Graphviz and Dynagraph – Static and dynamic graph drawing tools. In M. Junger and P. Mutzel, editors, *Graph Drawing Software*, Mathematics and Visualization, pages 127–148. Springer-Verlag, 2004. `doi:10.1007/978-3-642-18638-7_6`.

[8] E. Gansner, E. Koutsofios, S. North, and K.-P. Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, 1993. `doi:10.1109/32.221135`.

[9] R. Ierusalimschy. *Programming in Lua*. Lua.org, 2nd edition, 2006.

[10] S. Mauw and V. Bos. Drawing message sequence charts with LATEX. *TUGboat*, 22(1/2):87–92, 2001.

[11] J. Pohlmann. Configurable graph drawing algorithms for the TikZ graphics description language. Diploma thesis, Institute of Theoretical Computer Science, Universität zu Lübeck, Lübeck, Germany, Oct. 2011.

[12] E. M. Reingold and J. S. Tilford. Tidier drawings of trees. *IEEE Transactions on Software Engineering*, 7(2):223–228, 1981. `doi:10.1109/TSE.1981.234519`.

[13] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man and Cybernetics*, 11(2):109–125, 1981. `doi:10.1109/TSMC.1981.4308636`.

[14] T. Tantau. *The TikZ and PGF Packages, Manual for version 2.10-cvs*, 2013. Available online at http://sourceforge.net/projects/pgf/. Accessed March 2013.

[15] T. Tantau. Graph drawing in Ti*k*Z. In *Proceedings of Graph Drawing 2012*, volume 7704 of *Lecture Notes in Computer Science*, pages 517–528. Springer, 2013. doi:10.1007/978-3-642-36763-2_46.

[16] B. Veytsman and L. Akhmadeeva. Drawing medical pedigree trees with TEX and PSTricks. *TUGboat*, 28(1):100–109, 2007.