Graph Drawing in Ti*k*Z

Till Tantau

Institute for Theoretical Computer Science
Universität zu Lübeck
D-23562 Lübeck, Germany
tantau@tcs.uni-luebeck.de

Abstract. At the heart of every good graph drawing algorithm lies an efficient procedure for assigning canvas positions to a graph's nodes and the bend points of its edges. However, every 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 texts, and styling nodes and edges. We present a new framework, implemented in the Lua programming language and integrated into the Ti*k*Z graphics description language, that aims at simplifying the implementation of graph drawing algorithms. Implementers using the framework can focus on the core algorithmic ideas and will automatically profit from the framework's pre- and post-processing steps as well as from the extensive capabilities of the Ti*k*Z graphics language and the TEX typesetting engine. Algorithms already implemented using the framework include the Reingold-Tilford tree drawing algorithm, a modular version of Sugiyama's layered algorithm, and several force-based multilevel algorithms.

1 Introduction

A graph drawing algorithm is, mathematically speaking, 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 the text 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 (we will refer to users of graph drawing systems as "authors" in the following) must use to describe the graphs they wish to draw, 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 system outputs 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 mayor obstacle to implementing new graph drawing algorithms is that researchers are typically forced to address all three of the above problems, even

W. Didimo and M. Patrignani (Eds.): GD 2012, LNCS 7704, pp. 517–528, 2013.
© Springer-Verlag Berlin Heidelberg 2013

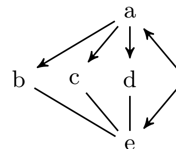
though they would like to focus on the implementation part. This leads to interesting graph drawing algorithms being available only as prototypes that lack many features necessary to make them usable in practice (an example is the force-based Lombardi graph drawer presented at this conference last year [3]). Even when algorithms are part of powerful toolkits, these toolkits may be difficult to integrate into a typesetting work flow (as is the case for MATHEMATICA or the Open Graph Drawing Framework [4]) 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 sometimes even when the same system is used).

The present paper introduces a new framework for implementing graph drawing algorithms that aims at letting researchers focus on the implementation part for new algorithmic ideas and that takes care of 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" [13], 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. The new graph drawing framework is tightly integrated into Ti*k*Z: in order to draw a graph using, say, the Sugiyama method [6,12], all an author has to do is to add the option "`layered layout`" to the description of a graph. Here is a typical excerpt from a manuscript (the output resulting from running TEX on it is shown on the right):

```
% TeX manuscript
Consider the diagram

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

Consider the diagram



While the framework makes it easy for authors to apply powerful graph drawing algorithms to graphs specified inside a TEX document, its main purpose 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 modern versions of TEX. Secondly, 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 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. [1]. 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.

Organisation of this Paper. After a short review of the graphics description language TikZ in Section 2, Section 3 discusses the syntax used in TikZ for the description of graphs. As will be argued, choices in the syntax are not only a matter of taste, but can influence the quality of graph drawings. In Section 4 we contrast the integration of a graph drawing framework into an existing graphics description language to the approaches taken by other graph drawing systems. Section 5 sketches the graph transformations performed by the framework and presents a complete implementation of a simple graph drawing algorithm. In the conclusion the future road map of graph drawing in TikZ is sketched.

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 [2] is an improved version of the Reingold–Tilford algorithm implemented directly in the TEX programming language. Since then, numerous independent packages have been developed in the context of TEX that implement a variety of specialized graph drawing algorithms covering different fields like drawing trees in linguistics or message sequence charts in software engineering.

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.

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 [4], a lot of the supporting libraries are not available and the implementation is much leaner. In both cases, the fact that the systems use compiled code makes them much faster than our Lua implementation.

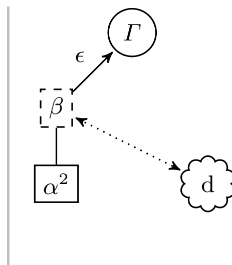
2 The Graphics Description Language TikZ

TikZ[1] 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 an extensive documentation. Being part of any standard TEX installation and being written entirely as a collection of macros, it can be used without further installation on any system running the TEX program. Its syntax borrows from METAFONT, a graphic description language designed by Donald Knuth, and PSTRICKS, a macro package similar to TikZ but tailored specifically to the POSTSCRIPT language that is understood by many printing devices. The macros of TikZ 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.

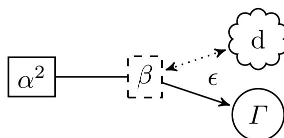
[1] TikZ means "TikZ ist *kein* Zeichenprogramm," a recursive German acronym in the tradition of "GNU is Not Unix" cautioning that TikZ is *not* a graphical editor.

The most important feature of Ti*k*Z for the purposes of the present paper is 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 following 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)
        (b) edge [->, "$\epsilon$"]    (c)
        (b) edge [<->, dotted]         (d);  };
```



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



3 On a Syntax for Specifying Graphs and the Quality of Graph Drawings

Graph drawing systems should make it easy for authors to specify the graphs 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.

Desirable Properties of Graph Description Languages for 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 through the means of the 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 [1]) 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 succinctly and in a 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 small compared to what is possible in TikZ; moreover, the set is not extensible. As a result, 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, orient=d|b]
  \graph {a -- b -- c -- d -- a; b -- d};
```

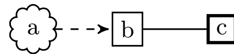
a d c b

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

a b c d δ_1 δ_2

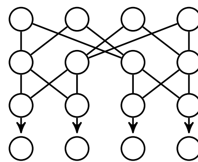
The first main difference to the DOT format is the place where options are specified: The `\graph` command, each node, and also each edge indicator (symbols like `--` or `->`) can be followed by options written in square brackets. In particular, in a sequence of nodes connected by edges, each node and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

```
\tikz [nodes=draw, tree layout, grow=right]
  \graph {a[cloud] ->[dashed] b -- c[very thick]};
```



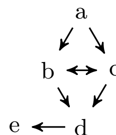
The second main difference concerns the semantics of edge indicators, of which there are five (`--`, `->`, `<-`, `<->`, and the special `-!-` meaning "remove an edge specified earlier"). While graph descriptions in DOT format are sequences of edges plus syntactic sugar (we can write `a -> b -> c` instead 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 and G_2 are two disjoint three-vertex graphs and γ is a *combining function* that combines two (possibly overlapping) graphs by taking their union and adding some edges. The default combining function connects the nodes using a matching, but it can be changed for each use of an edge indicator. The use of advanced combining functions allows authors to describe graphs in a succinct and, ideally, self-explaining manner as the following example of a butterfly network shows.

```
\tikz [nodes={draw, circle}, layered layout]
  \graph [empty nodes] {
    {a, b, c, d} --[butterfly={level=2}]
    {e, f, g, h} --[butterfly={level=1}]
    {i, j, k, l} -> {m, n, o, p} };
```



Options attached to nodes or edges provide "local" information to graph drawing algorithms. In order to communicate information about "global" 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 rank`" for same rank clusters. Using an option key like `same rank` 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 rank] b <-> c };
  {[same rank] d, e };
};
```



Special support is available (in the form of syntactic sugar) for specifying subgraphs on which a graph drawing algorithm should be used that differs from the main algorithm. 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 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 subgraph kind, an array stores all subgraphs that have been specified.

4 Integrating a Graph Drawing Framework into a Graphics Description Language

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. However, it also makes it necessary to establish an interface between the program employed for the typesetting and the program employed for the graph drawing, whenever documents should contain drawings of graphs.

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.

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) 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, it is

implemented as a subsystem of the typesetting program TEX, but using the Lua programming language. This language has been integrated into recent versions of TEX, making the implementation of advanced graph drawing algorithms 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.

The overall structure of the framework consists of three layers:

1. Document authors see the *TikZ layer.* Its job is to provide a powerful and easy-to-use syntax, such as the graph description syntax from the previous section. This layer is written in the TEX programming language.
2. Implementers of graph drawing algorithms see the *Lua layer* of the graph drawing framework. It consists of Lua classes for graphs, nodes, edges, as well as libraries for common tasks like algorithms for graph traversal, graph decomposition, and so on. This part of the framework can be used independently of TEX, making it possible to use the implemented algorithms in another program such as an interactive graph editor.
3. Between these two layers, there is a *binding layer* that passes the results of the parsing process from the Ti*k*Z layer down to the graph drawing algorithms and passes back the results (namely the computed coordinates of the nodes) to TEX. This layer is written in a mixture of TEX and Lua.

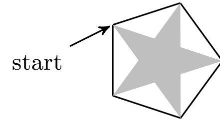
Let us consider `\tikz \graph [tree layout] {a -> b -> {c,d}};` as an example. Upon encountering the special option `tree layout`, Ti*k*Z starts a "graph drawing scope" inside which it will identify four nodes and three edges. Each of the four nodes is rendered normally, resulting in TEX boxes that contain 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 the binding layer intercepts the box at this point and passes it down to the Lua layer, where 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 the 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 on the Lua layer. At this point, the framework switches over completely to Lua and runs the graph drawing algorithm corresponding to the option `tree layout` (this happens to be the Reingold–Tilford algorithm). When the algorithm has finished, the node and edge objects will store the computed canvas coordinates. Since, from the typesetter's perspective, the nodes and edges "disappeared" during the parsing of the graph drawing scope, they must now be reinserted into the page. The binding layer inserts the stored nodes at the computed positions and, only now, also renders the edges.

Although the binding layer makes nodes disappear and reappear from the typesetter's memory, this is done in such a way that 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:

```
\tikz {
  \graph [spring layout, nodes=coordinate]
    {a -- b -- c -- d -- e -- a};
  \fill [black!25] (a)--(c)--(e)--(b)--(d);
  \node at(-1,-.5) {start} edge[->] (a); }
```



5 Implementing Graph Drawing Algorithms: Graph Drawing as a Sequence of Transformations

Graph drawing algorithms for Ti*k*Z are implemented on the Lua layer of the framework, using the Lua programming language. In the following, we first discuss the basic design principles of the Lua layer and then present a simple, but complete implementation of a tree drawing algorithm.

Graph Drawing as a Sequence of Transformations. The design of the graph drawing framework adheres to the philosophy advocated by Di Battista et al. [1] to see 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 the decomposition of the graph into connected components, or into bi- or tri-connected components, or planarization.

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 advertising the *algorithm properties.* For instance, a property "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 property "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 property "growth direction" tells the framework that the result of the transformation is embedded in the plane in such a way that certain further rotational transformations still need to be performed.

Properties 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, say, upwards. The framework will automatically rotate the graph when a user requests that the tree should grow in another direction and, having access to the convex hulls of the nodes, the framework can also correctly compensate for the effects of rotating a drawing of a graph, but not rotating the individual nodes.

Implementing a Graph Drawing Algorithm. We now present a simple, but complete example implementation of a tree drawing algorithm. It starts with a declaration of which kind of graphs it expects as input and what it will output:

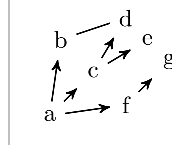
```
local TreeExample = pgf.gd.new_algorithm_class {
  works_only_on_connected_graphs = true,
  needs_a_spanning_tree = true,
  growth_direction = 90 } -- The algorithm "grows" the tree "upwards"
```

The declaration ensures that when the framework calls the `run` method (at the very end of the below code), the algorithm only gets a tree as input, even if the original input graph is not acyclic. (Like all other transformations, the to-be-used algorithm for computing spanning trees can be selected by authors through options.) The actual algorithm recursively positions subtrees next to each other, going right, and centers each node above its subtrees. (This placement strategy is not particularly clever and only used for demonstration purposes.)

```
local Coordinate = require "pgf.gd.model.Coordinate" -- An import
local function recursion(tree, vertex, left_end, y)
  -- The "options" table contains all options attached to the graph
  -- as a whole on the TeX level. Nodes and edges also have such tables.
  local level_dist     = tree.options['/graph drawing/level distance']
  local sibling_dist   = tree.options['/graph drawing/sibling distance']
  -- The array of edges starting at "vertex" in the graph "tree":
  local outgoing_edges = tree:outgoing(vertex)
  local right_end      = left_end
  for i,edge in ipairs(outgoing_edges) do -- Position the children
    right_end = recursion(tree, edge.head, right_end, y + level_dist)
    if i < #outgoing_edges then right_end = right_end + sibling_dist end
  end
  -- Now position the vertex, centered below its child trees
  vertex.pos = Coordinate.new ((left_end + right_end) / 2, y)
  return right_end
end
function TreeExample:run()
  recursion(self.spanning_tree, self.spanning_tree.storage.root, 0, 0)
end
return TreeExample -- The need for this return is a quirk of Lua
```

Putting the code in a file named `TreeExample.lua`, we can immediately use it as follows (note how the framework takes care of correctly handling options like the diagonal growth direction and does not pass the `b--d` edge to the algorithm):

```
% In the TeX manuscript:
\tikz [layout=TreeExample, grow'=45,
   sibling distance=1.25em, level distance=3em]
  \graph {a -> {b, c -> {d, e}, f -> g}, b -- d};
```



Naturally, the implementations of the "real" graph drawing algorithms that are already part of the framework are more complex than the above example: A not-quite-optimal implementation of the Reingold–Tilford [11] tree drawing algorithm has 130 lines of code; the modular implementation of the Sugiyama method [6,12] needs a bit over 2000 lines of code, nearly half of which implements the network simplex algorithm [5]; implementations of different force-based algorithms need between 350 and 500 lines of code and another 500 lines for supporting multilevel approaches.

6 Conclusion

Graph drawing in TikZ 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 graph drawing framework is implemented as a subsystem of a graphics description language, the full power of the language can be used to modify and augment graph drawings. Being part of the main TikZ code trunk, the current development version of the framework is available through `sourceforge.net/projects/pgf` and it 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 languages like C. None of the graphs in the present paper take any noticeable amount of time to process, but for instance a 47-node graph from the Graphviz example suite, depicting the Unix history, needs about four seconds. Most time is actually needed by the TikZ parser, the Lua part needs 1.2 seconds on a 2.8GHz CPU; but larger graphs may need minutes or hours to process. This problem can partly be sidestepped by using TikZ's built-in facilities for only (re)processing pictures whose syntactic description has changed in the manuscript. 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.

The development road map for graph drawing in TikZ is, firstly, to implement a larger range of standard graph drawing algorithms as well as further pre- and post-transformations such as computing SPQR tree decompositions. Second, I intend to refactor the binding layer's code to make bindings easy between the Lua part (which includes the implemented graph drawing algorithms) and systems other than TeX, such as interactive graph editors.

Acknowledgements. The first prototype of the framework was implemented as a graduate project by Ahrens, Frahm, Kluttig, Schulz, and Schuster [13, Chapter 31]. The prototype was greatly extended by Jannis Pohlmann while working on his Diploma thesis [10], where he details his implementations of the modular Sugiyama algorithm and of the multilevel force-based methods. I would also like to thank the anonymous reviewers for their valuable feedback.

References

1. Battista, G.D., Eades, P., Tamassia, R., Tollis, I.G.: Graph Drawing, Algorithms for the Visualization of Graphs. Prentice Hall (1999)
2. Brüggemann-Klein, A., Wood, D.: Drawing trees nicely with TEX. Electronic Publishing 2(2), 101–115 (1989)
3. Chernobelskiy, R., Cunningham, K.I., Goodrich, M.T., Kobourov, S.G., Trott, L.: Force-Directed Lombardi-Style Graph Drawing. In: van Kreveld, M., Speckmann, B. (eds.) GD 2011. LNCS, vol. 7034, pp. 320–331. Springer, Heidelberg (2012)
4. Chimani, M., Gutwenger, C., Jünger, M., Klein, K., Mutzel, P., Schulz, M.: The open graph drawing framework. Poster at the 15th International Symposium on Graph Drawing 2007 (GD 2007) (2007)
5. Cunningham, W.H.: A network simplex method. Mathematical Programming 11(1), 105–116 (1976)
6. Eades, P., Sugiyama, K.: How to draw a directed graph. Journal of Information Processing 13(4), 424–436 (1990)
7. Ellson, J., Gansner, E., Koutsofios, E., North, S., Woodhull, G.: Graphviz and dynagraph – static and dynamic graph drawing tools. In: Junger, M., Mutzel, P. (eds.) Graph Drawing Software. Mathematics and Visualization, pp. 127–148. Springer (2004)
8. Gansner, E., Koutsofios, E., North, S., Vo, K.P.: A technique for drawing directed graphs. IEEE Transactions on Software Engineering 19(3), 214–230 (1993)
9. Ierusalimschy, R.: Programming in Lua. Lua.org, 2nd edn. (2006)
10. Pohlmann, J.: Configurable Graph Drawing Algorithms for the Ti*k*Z Graphics Description Language. Diploma thesis, Institute of Theoretical Computer Science, Universität zu Lübeck (October 2011)
11. Reingold, E.M., Tilford, J.S.: Tidier drawings of trees. IEEE Transactions on Software Engineering 7(2), 223–228 (1981)
12. Sugiyama, K., Tagawa, S., Toda, M.: Effective representations of hierarchical structures. Tech. Rep. 8, International Institute for Advanced Study of Social Information Science, Fujitsu (1979)
13. Tantau, T.: The Ti*k*Z and PGF Packages, Manual for version 2.10-cvs (2012), `http://sourceforge.net/projects/pgf/`