Notes on an implementation of Sugiyama's scheme

Lev Nachmanson

Microsoft Research, One Microsoft Way, Redmond, WA, USA
levnach@microsoft.com

Abstract. These technical notes are comments on my work on GLEE, Graph Layout Execution Engine. GLEE is a library developed inside Microsoft Research and is written in the C# language. The library has quite a few users inside of Microsoft. GLEE implements Sugiyama's scheme of layout for directed graphs, or so-called layered layout. The paper describes in more detail efficient coding of adjacent vertex swaps and spline routing, where the latter is based on the b-spline in a channel approach by Lutterkort and Peters.

1 Introduction

Eiglsperger et al [4] mention that most practical implementations of graph layout engines follow Sugiyama's scheme, and GLEE is no exception. The scheme consists of several consecutive steps [5]. These steps are: cycle removal, layering, vertex ordering, horizontal coordinate assignment, and, finally, spline routing.

1) At the cycle removal step we transform the given directed graph into a DAG by reverting some edges and thus breaking the cycles.

2) In the layering step we assign each vertex to some layer (a horizontal row), in a way that each edge goes down from a higher layer to a lower one. Now we are almost ready for the ordering step, but we make an intermediate step where we create a *layered graph*. We introduce dummy vertices and replace each edge crossing more than two layers by a sequence of connected edges where each edge goes only one layer down. In this way we replace the DAG with a layered graph.

3) During the ordering step vertices are swapped within the layers to reduce edge crossings.

4) In the next step we fix x-coordinates of the layered graph's vertices by shifting (but not swapping), them within the layers.

5) The last step is spline routing where for each edge we create a spline, trim it with the source and target node boundaries, and calculate the arrow head positions.

The main result of the paper is a new algorithm for counting edge crossings in the adjacent swap phase, a sub-step of the ordering step. This is discussed in section 3. The description of the spline routing step implementation can be found in section 4. I touch on some issues concerning layering in section 2. Future work is discussed in section 6. All the sections can be read independently.

2 Layering

In spite of the fact that the layering step in GLEE is implemented according to [5], I'd like to make some remarks about the step. This section shows why the recipe given in [5] works. I also point to a possibly better method for solving the layering problem.

Let V be the set of the DAG vertices, and E be the set of its edges. Every edge e has weight $W[e]$ and separation $S[e]$. Both vectors W and S have nonnegative integer values. The weight of an edge defines the importance of keeping the edge short. The separation means the minimal possible span of an edge in the y-direction. For a correct layering every edge e has to cross at least $S(e) + 1$ layers. A layering is an integer valued vector y defined on V providing for any $v \in V$ its layer $y[v]$. We are required to find a layering y, such that $y[u] - y[v] \geq S(u, v)$ for each $(u, v) \in E$ and $\sum_{(u,v) \in E} W(u, v)(y[u] - y[v])$ is minimal.

It is useful to reformulate the layering problem as a linear program. Let A be the vertex-edge adjacency matrix of the DAG. The columns of A correspond to the edges and the rows correspond to the vertices. For every $i \in V$ and $(u, v) \in E$

$$A[i, (u, v)] = \begin{cases} 1, & \text{if } i = u \\ -1, & \text{if } i = v \\ 0, & \text{otherwise.} \end{cases} \tag{1}$$

Let us denote by D the vector defined on V, such that

$$D[u] = \sum_v W(u, v) - \sum_v W(v, u) \tag{2}$$

for every $u \in V$. The layering task can be stated in these terms as the following linear program: minimize Dy under conditions $yA \geq S$ and $y \in R^{|V|}$. The dual of this program can be stated as the following: minimize $-Sx$ under conditions $Ax = D$, $x \in R_+^{|V|}$, where R_+ is the set of nonnegative numbers.

One can see that the last program is the Transshipment Problem [3]. Usually problems of this kind are solved, as it's done in [5], by the Network Simplex method. Gansner et al [5] knew about the connection, but, to my knowledge, have not mentioned it in their publications.

The layering step usually takes less than 5% of the total running time. In my opinion, the version of the algorithm from [3] is easier to implement and may work even faster.

3 Efficient counting of edge crossings during adjacent swaps

The ordering step starts when we have a layered graph, but the order of vertices within a single layer is not yet defined. We traverse the layers up and down several times applying the median method of [5], and create some ordering within the layers. Counting the crossings of edges connecting two neighboring layers at this stage is done by using the technique from [1]. The next sub-step of the ordering step is the swapping of vertices which are adjacent on the same layer. Gansner et al [5] point out that the adjacent vertex swaps reduce the number of edge crossings by 20-50% thus improving the layout quality. If one is not careful enough, counting of edge crossings at this phase could become

a bottleneck of the layout calculation. The approach and data structures suggested here lead to an efficient implementation.

Proposition 1 *Swap of vertices u and v can be produced with the amortized cost $O(d(u) + d(v))$, where d is the degree of layered graph vertices.*

By the cost of a swap we mean the cost of calculating the change in the number of the edge crossings after the swap and the updates for the related data structures.

Let us show how to achieve proposition 1. To do that we need to define some data structures.

If (u, v) is an edge of the layered graph, then we call u a predecessor of v, and v a successor of u. For every vertex v, let $P(v)$ be a sequence of predecessors of v, and $S(v)$ be a sequence of successors of v. Because of the fact that the graph is layered, all vertices of $P(v)$ belong to the same layer. It is also true for $S(v)$. We will keep elements of each $S(v)$ and $P(v)$ ordered according to the orders induced by the layers containing them.

For each vertex v let *Po*(v) be a function from $P(v)$ to the set of integers, such that for any $u \in P(v)$ value *Po*$(v)(u)$ is the offset of u in $P(v)$. In other words, $P(v)[Po(v)(u)] = u$. Similarly, we define *So*, such that $S(v)[So(v)(u)] = u$ holds for every vertex v and every $u \in S(v)$. In addition, let X be an integer-valued function defined on the set of graph vertices giving the horizontal position of vertices within the layers; that is, if L is a layer and $v \in L$, then $L[X(v)] = v$. In figure 3 $Po(u) = \{a \rightarrow$

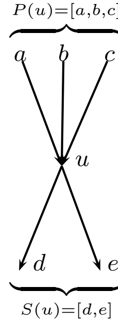

Fig. 1. Sequences $S(u)$ and $P(u)$

$0, b \rightarrow 1, c \rightarrow 2\}$ and $So(u) = \{d \rightarrow 0, e \rightarrow 1\}$. This construction helps us to avoid unnecessary sorting and is easy to update.

Suppose that vertices u and v are adjacent on a layer, and we are considering them for a possible swap. Consider an edge e which is adjacent neither to u nor to v. Swapping u and v does not change the number of crossings of e with the rest of the edges. Therefore, to decide if the swap is beneficial, we only take into account intersections between edges adjacent to u or to v. Let *cuv* (*cvu*) be the number of intersections between edges adjacent to u or v when u is to the left(right) of v in the layer. If u is to

the left of v, then the swap is beneficial only when *cuv* $>$ *cvu*. Our task is to find cuv and cvu efficiently. Number *cuv* can be represented as the sum of values *upperCuv* and *lowerCuv*, where *upperCuv* is the number of crossings of edges incoming into u or v, and *lowerCuv* is the number of crossings of edges outgoing from u or v.

We show how to find *upperCuv*; *lowerCuv* is calculated similarly. Let $P(u)$ be $[a_1, ..., a_n]$, and $P(v)$ be $[b_1, ..., b_m]$. The sequences $A = [X(a_1), ..., X(a_n)]$ and $B = [X(b_1), ..., X(b_m)]$ represent horizontal positions of vertices of $P(u)$ and $P(v)$ correspondingly. Note that A and B are increasing sequences since they follow the order of the layer above v. As shown in [1], *upperCuv* is equal to the number of inversions between A and B: that is, the number of pairs i,j, such that $X[a_i] > X[b_j]$. For the sake of completeness, we provide the procedure calculating this value.

```
UCUV() {
  ucuv=0;
  inversions=n;
  i=1,j=1;
  while(i ≤ n and j ≤ m)
     if(X[b[j]]-X[a[i]] ≥ 0){
        i=i+1;
        inversions=inversions-1;
     }
     else {
        ucuv=ucuv+inversions;
        j=j+1;
     }
}
```

Lemma 1. *Procedure $UCUV$ works $O(m + n)$ steps.*

Proof. In each cycle or i increases, or j increases.

After swapping u and v we need to update the corresponding structures $P, S,$ *Po* and *So*. The following observation allows us to do it efficiently:

Lemma 2. *The swapping of u and v changes $P(w)$ ($S(w)$) if and only if w is a common successor(predecessor) of u and v.*

Proof. Indeed, the swap of u and v changes some $P(w)$ if and only if $u, v \in P(w)$. If vertex $l \in P(w)$ then, by definition, w is a successor of l. The other part of the lemma is proved similarly.

The procedure below updates S and *So* after swapping u and v when u is located to the left of v. Structures P and *Po* are treated similarly.

```
foreach (w ∈ P(u)){
  let s=S(w);
  let r=So(w);
  if( v ∈ domain(r){
```

```
    //here we know that w is a common predecessor of u and v
            let vOffset=r(v);
            s(vOffset-1)=v;
            s(vOffset)=u;
            r(v)=vOffset-1;
            r(u)=vOffset;
    }
}
```

If we implement $So(w)$ as a hash table for every w, then the amortized cost of the routine above is $|P(u)|$. Indeed, the loop itself works $|P(u)|$ times, and the query $v \in domain(r)$ has amortized cost $O(1)$ for a hash table r. The updating of X and the layer can be done in $O(1)$ since we know the old and the new positions of u and v. The amortized cost of all required updates after swapping u, v is then $O(\min(|S(u)|, |S(v)|) + \min(|P(u)|, |P(v)|))$. The minimum comes from the fact that we could have started the loop above from $P(v)$ if it had fewer elements than $P(u)$. That, together with Lemma 1, proves Proposition 1.

We have avoided sorting while doing adjacent swaps, and, in fact, we can avoid it completely. The initialization step can also be done without sorting, since we can fill P and S by walking over the layers in their order and adding a new element at the first unoccupied position of the array.

One can achieve the same performance bound as stated in Proposition 1 by using the radix sort; however, as my experiments show, the suggested method is about 3-5 times faster.

4 Spline routing

In the heart of the spline routing in GLEE lies the method of [6]. Here we explain the way we apply this method. The idea of [6] is to build a channel, an area bounded by two polylines, and create a b-spline fitting into the channel. More formally, the boundaries of the channel are given by real-valued, piecewise linear functions l and h. The both functions are defined on the same domain, a segment $[a, b]$ where $a < b$. The curve l is the low boundary and h is the upper boundary of the channel: for every $x \in [a, b]$ the inequality $l(x) \leq h(x)$ holds. See figure 4 for a channel sample. A curve b defined on $[a, b]$ fits into the channel if for every $x \in [a, b]$ we have $l(x) \leq b(x) \leq h(x)$. When we find b fitting into the channel, we create a curve bp in the plane R^2 which is also defined on the segment $[a, b]$, and $bp(x) = (b(x), x)$ for $x \in [a, b]$. We are looking for a b-spline of the third degree that can be conveniently represented as a sequence of cubic Bezier segments and rendered by a graphics library.

We need to give some minimal information about b-splines, just enough for sketching of how GLEE deals with routing. A b-spline is completely defined by two sequences of real numbers: knots and control points. One can think about knots as of a partition of the spline domain. We need a notion of a Greville abscissa. For a spline of the degree d and the knot sequence $[t_i]$ Greville abscissas are numbers of the form $1/d \sum_{i=k+1}^{i+d} t_i$. As shown in [6], for an increasing sequence g of real numbers one can find a sequence of knots, such that their Greville abscissas contain g.

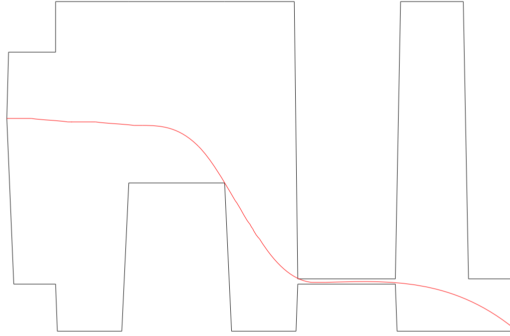

Fig. 2. A channel with a spline

Now we are ready to describe the algorithm. The core of the algorithm is in finding a spline by a given knot sequence, l and h. It is done as in [6], by solving a linear program. Let's call this linear program $L(k, l, h)$, where k is the knot sequence. The solution of the program gives us a sequence of control points and we construct the spline from the knots and the control points. The program constraints force the spline to fit into the channel, and by minimizing the cost we make the spline straight. Often $L(k, l, h)$ is infeasible. This brings us to an iterative process:

```
Create an initial knot sequence k.
do{ Try to solve L(k,l,h).
   If no solution exists, refine k. }
while(there is no solution)
```

We try to keep the knot sequence short as we iterate. There are two reasons for doing it.

a) Usually splines with small number of knots are more aesthetic than the ones with large number of knots.

b) The size of $L(k, l, h)$ grows quadratically with the length of k and the program takes much longer time to solve.

Intuitively, Greville abscissas are spline parameters where the knots have the most influence on the spline behavior. While iterating we keep around a vector g of parameters where we would like to constrain the spline. All values of g belong to the segment $[a, b]$. The knot sequence is calculated based on g, as mentioned above, in a way that all components of g are Greville abscissas of the knots. Let's denote by $k(g)$ such a sequence. The initial value of g is $\{g[0] = a, g[1] = b\}$. If $L(k(g), l, h)$ does not have a solution we refine g. To find new members of g we create a spline which is in an approximation of the solution of $L(k(g), l, h)$, cross this spline with l and h, and insert into g the spline parameters corresponding to the intersection points. To keep g short we fix some number $d > 0$ and avoid the insertion when the distance between two members of g becomes smaller than d. This measure also bounds the growth of the knot sequence. In the current implementation $d = (b - a)/240$.

Let's explain how we find an approximate solution of an infeasible $L(k(g), l, h)$. The linear program can be represented in the standard form: minimize cx under condition $Ax = b, x \geq 0$, where A is a matrix and x,b and c are vectors. For an approximate solution we take $x^* \geq 0$ minimizing $||Ax - b||$ for $x \geq 0$. Such an x^* is the closest to b point of the set $\{Ax : x \geq 0\}$. Problems of this kind are called quadratic programs, and they can be solved by most of solvers. A simple to implement but not a very efficient approach to this problem can be found in [9].

Because of the limitations of the current linear program solver in GLEE we divide the problem in two when the matrix A becomes too large. We split the channel in its narrowest place and thus create two new tasks. Usually it produces an undesirable effect on the spline which is now a concatenation of two splines and is not optimized. Luckily, the division does not happen too often.

5 Samples of graph layouts created by GLEE

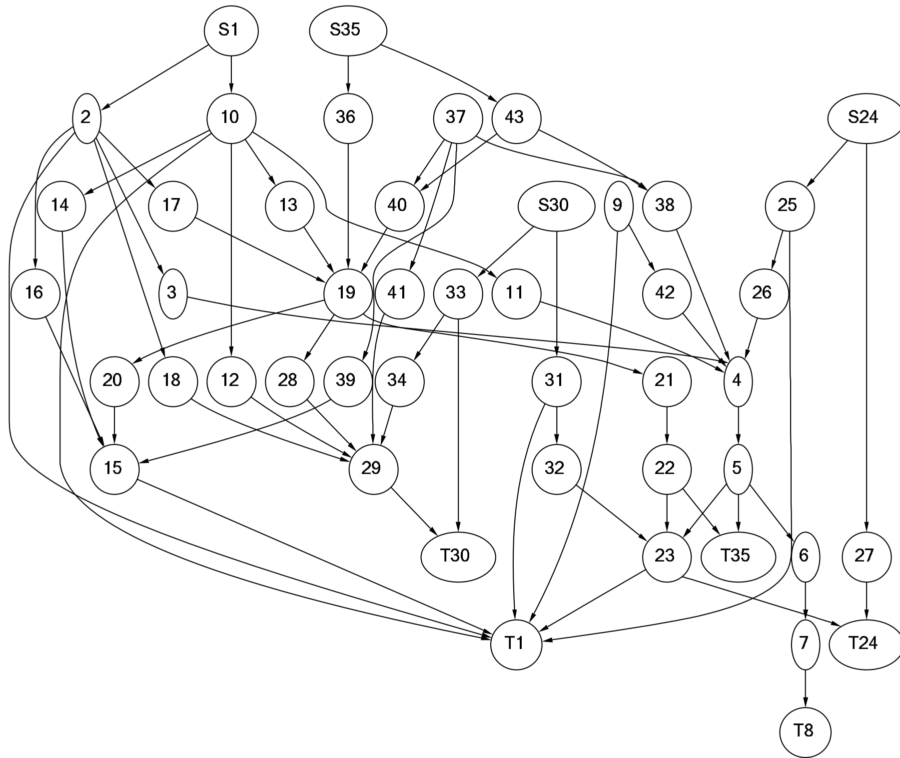

Fig. 3. World dynamic model [8]

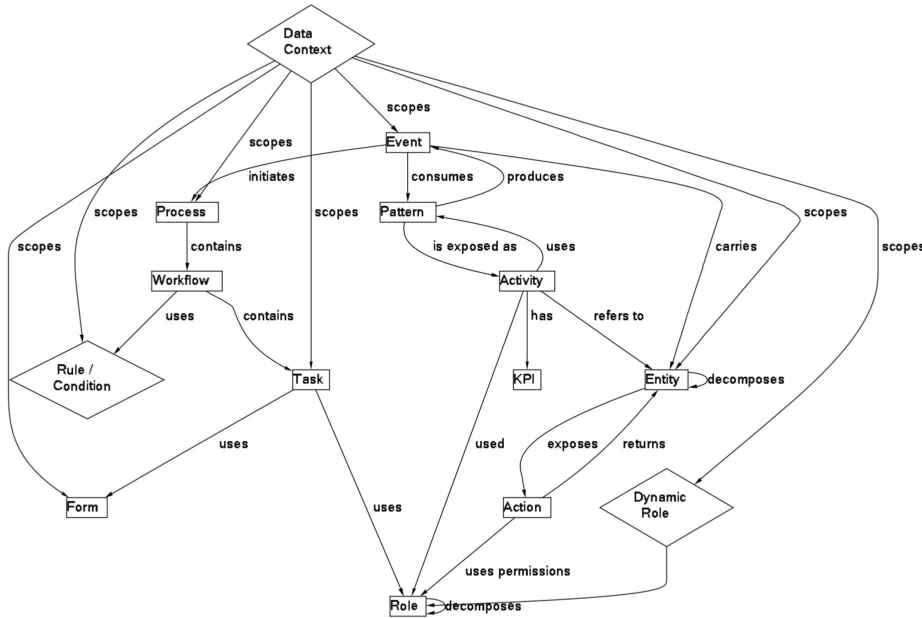

Fig. 4. An hierarchy

The graph displayed in figure 3 comes from [8]. Figure 4 exposes a complex hierarchy of relations between entities. The graph at figure 5 is the control flow graph of a program.

6 Future work

Inspite of the fact that GLEE is used successefully by several Microsoft groups, there are several aspects of GLEE that need improvement. Among them are performance, the quality of the splines and support for different constraints on the layout.

The performance bottleneck for GLEE is the step of assigning horizontal coordinates. This step is done by reducing the problem to the layering and then reusing the Network Simplex method to solve it. There are two papers [2, 4] suggesting more efficient approaches. The authors of [2] found out a linear time method of assigning horizontal coordinates. From the other side, [4] concentrates on introducing fewer additional dummy vertices thus making the size of the problem of the total layout solvable in $O(|V|+|E|)log(E)$ time while requiring $O(|V|+|E|)$ space. GLEE will be adapted to use these approaches. The approach of [2, 4] will introduce more horizontal balance into GLEE layouts.

Splines of GLEE have the limitation of being graphs of functions from the real line y to the real line x. In other words, when layout is done top to bottom, a GLEE spline intersects a horizontal line at most at one point. This restriction forces GLEE to leave a gap between the lowest bottom of a node on a layer and the highest top of a node on the

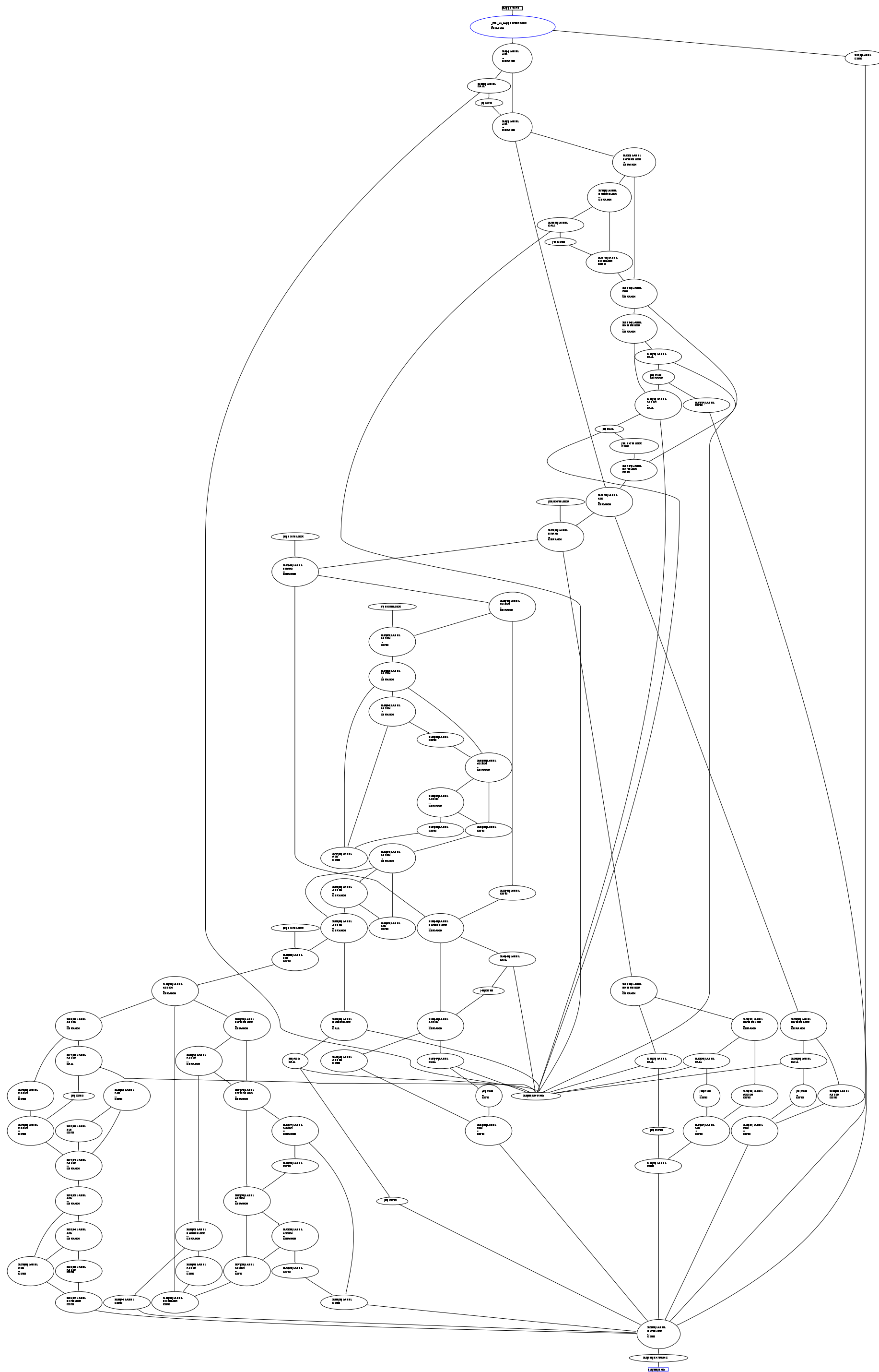

Fig. 5. A control flow graph

layer below. Sometimes it creates graphs which are unnecessarily tall. The limitation can be removed by using the technique from [7] where the authors show how to fit splines into “winding” channels.

References

1. W. Barth, M. Jünger, and P. Mutzel. Simple and efficient bilayer cross counting. In M. T. Goodricht and S. G. Kobourov, editors, *Graph drawing: 10th International Symposium, GD 2002, Irvine, CA, USA, August 2002, Revised Papers*, volume 2528 of *Lecture Notes in Computer Science*, pages 130–141, New York, NY, USA, 2002. Springer-Verlag Inc.
2. U. Brandes and B. Köpf. Fast and simple horizontal coordinate assignment. In P. Mutzel, M. Jünger, and S. Leipert, editors, *Graph drawing: 9th International Symposium, GD 2001, Vienna, Austria, September 2001, Revised Papers*, volume 2265 of *Lecture Notes in Computer Science*, pages 31–44, New York, NY, USA, 2002. Springer-Verlag Inc.
3. V. Chvatal. *Linear Programming*. A Series of Books in the Mathematical Sciences. Freeman, 1983. ChVA v 83:1 P-Ex.
4. M. Eiglsperger, M. Siebenhaller, and M. Kaufmann. An efficient implementation of sugiyama’s algorithm for layered graph drawing. In J. Pach, editor, *Graph Drawing, New York, 2004*, pages pp. 155–166. Springer, 2004.
5. E. R. Gansner, E. Koutsofios, S. C. North, and K.-P. Vo. A Technique for Drawing Directed Graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, Mar. 1993.
6. D. Lutterkort and J. Peters. Smooth paths in a polygonal channel. In *Symposium on Computational Geometry*, pages 316–321, 1999.
7. A. Myles and J. Peters. Threading splines through 3d channels. *Computer-Aided Design*, 37(2):139–148, 2005.
8. L. A. Rowe, M. Davis, E. Messinger, C. Mayer, C. Spirakis, and A. Tuan. A Browser for Directed Graphs. *Software – Practice and Experience*, 17(1):61–76, Jan. 1987.
9. P. Wolfe. The simplex method for quadratic programming. *Econometrica*, 1959.