Let's Draw a Graph: An Introduction with Graphviz

Marc Khoury

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2013-176
http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-176.html

October 28, 2013

Let's Draw a Graph

An Introduction with Graphviz

Marc Khoury

1 Introduction

Graphs are ubiquitous data structures in computer science. Many important problems have solutions hidden in the complexity of modern graphs, rendering effective visualization techniques extremely valuable. The need for such visualization techniques has led to the creation of a myriad of graph drawing algorithms.

We present several algorithms to draw several of the most common types of graphs. We will provide instruction in the use of Graphviz, a popular open-source graph drawing package developed at AT&T Labs, to execute these algorithms. All figures shown herein were generated with Graphviz.

2 The DOT Language

Visualization of a given graph requires that it first be represented in a format understandable by graph drawing packages. We will use the DOT format, a format that can encode most attributes of a graph in a human-readable manner [1].

2.1 Undirected Graphs

A DOT file for an undirected graph begins with the keyword *graph* followed by the name of the graph. An undirected edge between vertices u and v can be specified by u`--`v. A simple example of an undirected graph with five vertices and five edges is illustrated by Figure 1, below.

Listing 1: A DOT file for a simple undirected graph with five vertices.

```
graph graphname {
    1 -- 2;
    3 -- 2;
    4 -- 1;
    2 -- 5 -- 4;
}
```

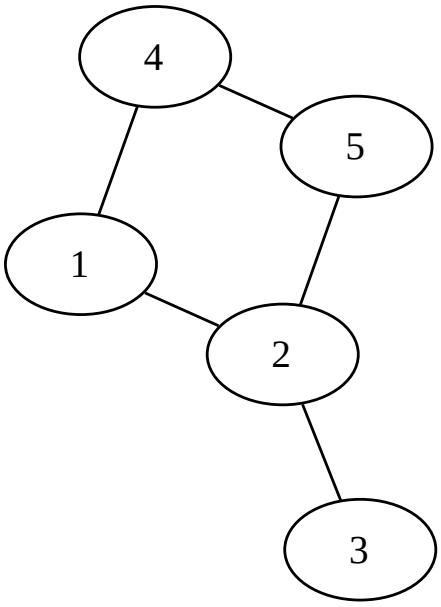


Figure 1: Visualization of Listing 1 graph.

The Graphviz application *neato* is a straightforward method of rapidly visualizing undirected graphs in the format described above. Figure 1 was generated by the command `neato -Teps undirected.gv > undirected.eps`, where undirected.gv is a file containing the code shown in Figure 1 and -Teps specifies an Encapsulated Postscript output format. Graphviz supports a wide range of output formats including GIF, JPEG, PNG, EPS, PS, and SVG. All Graphviz programs perform I/O operations on standard input and output in the absence of specified files.

2.2 Directed Graphs

A directed graph begins with the keyword *digraph* followed by the name of the graph. A directed edge between two vertices u and v is specified by u`->`v. The aforementioned edge starts at u and goes to v. The DOT code for and visualization of an example directed graph appears in Listing 2 and Figure 2, respectively. The visualization in Figure 2 was produced via the command `dot -Teps directed.gv > directed.eps`.

Listing 2: A DOT file for a simple directed graph.

```
digraph graphname {
     a -> b;
     a -> c -> d;
     c -> e;
}
```

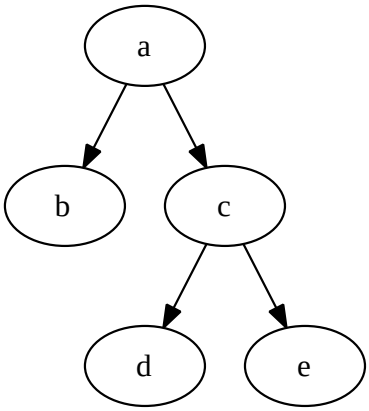


Figure 2: Visualization of Listing 2 graph.

2.3 Attributes

The DOT format supports a wide selection of attributes for vertices, edges, and graphs. Examples of user-definable attributes include the color and shape of a vertex or the weight and style of an edge. There are far too many attributes to list here and we direct the reader to the Graphviz documentation for a comprehensive list. Graphviz defines default values for most of the attributes available for DOT files. Many attributes are only used by specific Graphviz programs. As an example, the repulsive force attribute is only used by the *sfdp* module.

Listing 3: A DOT file where most vertices and edges have been assigned various attributes.

```
graph graphname {
     a [label="Root", shape=circle];
     b [shape=box, color=red];
     a -- b -- c [color=blue];
     b -- d [style=dotted];
     a -- e -- f [color=green];
     f [label="Leaf"];
 }
```

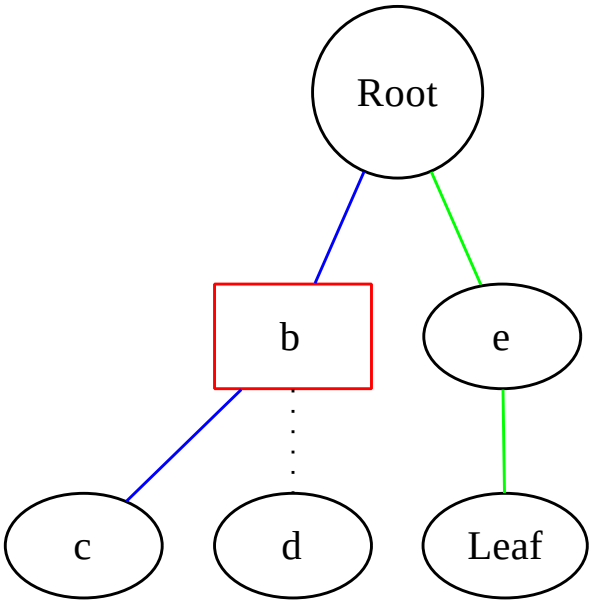


Figure 3: Visualization of Listing 3 graph.

Attributes may be used to draw attention to sections of a graph (e.g. color and thickness adjustment to highlight a path).

Listing 4: A DOT file where edges along a path have been colored red.

```
graph {
    2--9--1--8--7 [color=red,penwidth=3.0];
    6--9
    6--5
    1--3
    7--5
    3--2
    0--4
    4--9
    0--6
    5--8
    6--4
}
```

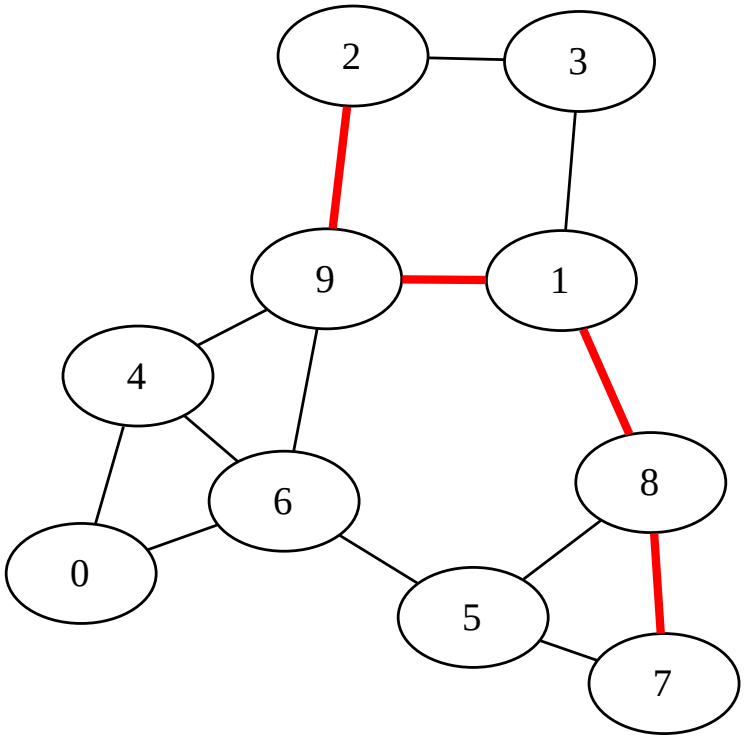


Figure 4: Visualization of Listing 4 graph.

2.4 Clustering

In some graphs, grouping of vertex subsets is desirable. This grouping of vertex subsets is particularly useful in the case of k-partite graphs. The DOT language allows the specification of subgraphs which can be clustered together and visually separated from other parts of the graph.

In a graph file the keyword *subgraph* is used to specify a subgraph. Prefixing the name of the subgraph with the expression "cluster_" ensures that the subgraph will be visually separated in the layout. Note that only *dot* and *fdp* (explained in detail later in this document) support clustering. Figure 5 was generated by the command `dot -Teps cluster.gv > cluster.eps`.

Listing 5: A graph with two clusters representing different processes.

```
digraph cluster{

     subgraph cluster_0{
          label="Process A";
          node[style=filled, color="lightgray"];
          a0 -> a1 -> a2 -> a3;
     }
     subgraph cluster_1 {
          label="Process B";
          b0 -> b1 -> b2;
     }
     b1 -> a3;
     start -> a0;
     start -> b0;
     a3 -> end;
```

```
    b2 -> end;
    start [shape=Mdiamond];
    end [shape=Msquare];
}
```

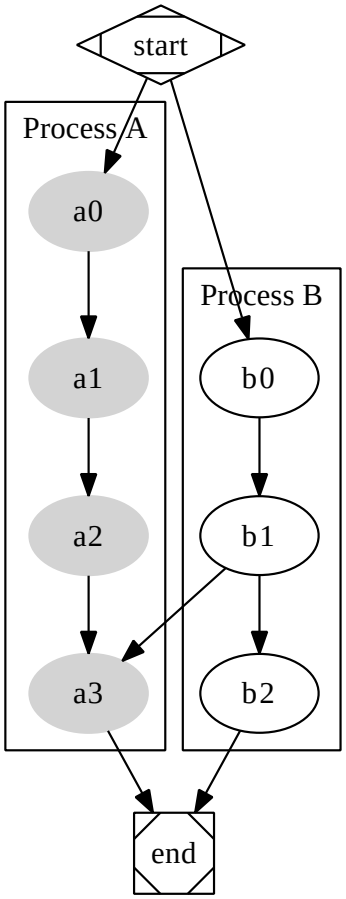


Figure 5: Visualization of Listing 5 graph.

3 Force-Directed Methods

Force-directed algorithms model graph layouts as physical systems, assigning attractive and repulsive forces between vertices and minimizing the total energy in the system. The optimal layout is defined as the layout corresponding to the global minimum energy.

The spring-electrical model assigns two forces between vertices: the repulsive force f_r and the attractive force f_a. The repulsive force is defined for all pairwise combinations of vertices and is inversely proportional to the distance between them. The attractive force exists only between neighboring vertices and is proportional to the square of the distance. Intuitively, every vertex wants to keep its neighbors close while pushing all other vertices away [7].

$$f_r(i,j) = \frac{-CK^2}{\|x_i - x_j\|}, i \neq j, i, j \in V \tag{1}$$

$$f_a(i,j) = \frac{\|x_i - x_j\|^2}{K}, (i,j) \in E \tag{2}$$

The force on vertex i is the sum of the attractive (from vertices $j \in V$, $j \neq i, (i,j) \in E$) and repulsive (from vertices $j \in V$) forces.

$$f(i,K,C) = \sum_{i \neq j} \frac{-CK^2}{\|x_i - x_j\|}(x_j - x_i) + \sum_{(i,j) \in E} \frac{\|x_i - x_j\|^2}{K}(x_j - x_i) \tag{3}$$

The parameter K is known as the optimal distance and the parameter C is used to control the relative strength of the attractive and repulsive forces. While the choice of these parameters is important in practice, mathematically it can be shown that they only scale the layout [7].

Finally, the total energy in the system is the sum of the squared forces.

$$\text{energy}(K,C) = \sum_{i \in V} f(i,K,C)^2 \tag{4}$$

For each vertex i the algorithm computes the forces acting on i and adjusts the position of i in the layout. The algorithm repeats this process until it converges on a final layout. The presented force-directed layout algorithm requires $O(V^2)$ time per iteration and it is generally considered that $O(V)$ iterations are required.

Algorithm 1 ForceDirectedLayout(G,x,tol)

step $\leftarrow$ initial step length
while not converged **do**
  $x^0 \leftarrow x$
  for $i \in V$ **do**
    $f \leftarrow 0$
    for $(i,j) \in E$ **do**
      $f \leftarrow f + \frac{f_a(i,j)}{\|x_i - x_j\|}(x_j - x_i)$
    end for
    for $j \neq i$, $j \in V$ **do**
      $f \leftarrow f + \frac{f_r(i,j)}{\|x_i - x_j\|}(x_j - x_i)$
    end for
    $x_i \leftarrow x_i$+step$*f/\|f\|$
  end for
  step $\leftarrow 0.9*$ step
  if $\|x - x^0\| < K*$tol **then**
    converged $\leftarrow true$
  end if
end while
return x

The Graphviz program *fdp* uses a force-directed algorithm to generate layouts for undirected graphs. *fdp* is suitable for small graphs that are both unweighted and undirected. Figure 6, below, displays a layout generated by *fdp* for a torus generated using the command `fdp -Teps torus.gv > torus.eps`.

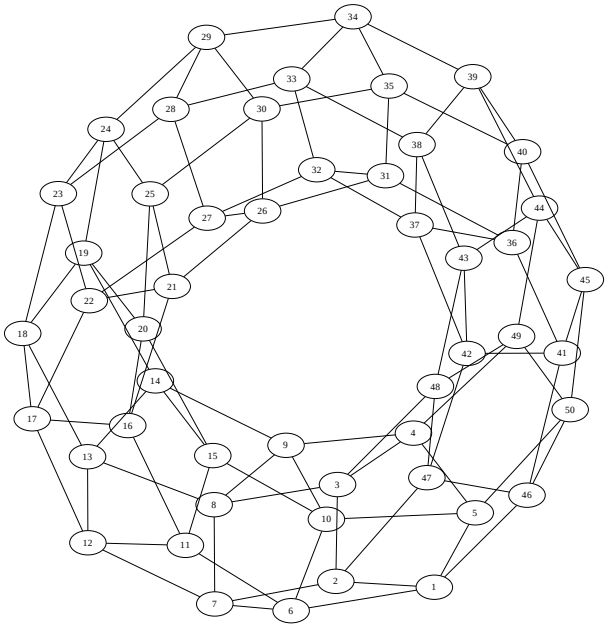


Figure 6: A drawing of a torus produced by fdp.

The complexity of this computation can be decreased by employing a Barnes-Hut scheme to compute the forces acting on a vertex [2]. The use of a quadtree to group vertices into supernodes allows a single computation to approximate the force contributions of a large collection of vertices. This scheme reduces the complexity of the innermost loop from $O(V)$ to $O(\log(V))$, dropping the whole algorithm to $O(V \log(V))$ per iteration. The speed of this technique can be further improved by using a multilevel approach that takes advantage of graph coarsening techniques. The Graphviz program *sfdp* implements these techniques and is currently the optimal choice for generating large graph layouts.

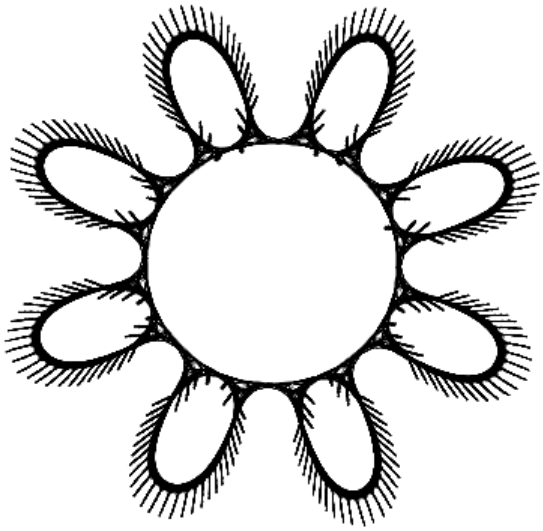


Figure 7: A drawing of a graph with 40000 vertices produced by sfdp.

4 Stress Majorization

Stress majorization attempts to embed the graph metric into $\mathbb{R}^k$ [5]. If the shortest path distance between two vertices $i, j \in V$ is d_{ij}, then stress majorization will attempt to place these two vertices at distance d_{ij} apart in the layout. To accomplish this, stress majorization uses an iterative optimization process to find a global minimum of the stress function shown in Equation 5. The iterative equation involves two weighted Laplacian matrices. For a quick introduction to Laplacian matrices and their properties, please see Appendix A.

Laplacian matrices permit many different types of weightings. Here we consider two weighted Laplacian matrices that are important for stress majorization.

Let d_{ij} be the shortest path distance - sometimes referred to as the "graph-theoretic distance" - between two vertices $i, j \in V$. Let $w_{ij} = d_{ij}^{-p}$ and choose $p = 2$. Technically, p could be any integer, but $p = 2$ seems to produce the best graph drawings in practice.

Definition 1 *Let G be a graph and let d be the shortest path distance matrix of G. Define $w_{ij} = d_{ij}^{-2}$. The weighted Laplacian L^w of a graph G is a $n \times n$ matrix given by:*

$$L^w_{i,j} = \begin{cases} -w_{ij} & \text{if } i \neq j \\ \sum_{k \neq i} w_{ik} & \text{if } i = j \end{cases}$$

The off diagonal elements of the weighted Laplacian are $-w_{ij}$, as opposed to -1 and 0. The diagonal element $L^w_{i,i}$ is the positive sum of the off diagonal elements for row i, as with the standard Laplacian matrix.

The second type of weighted Laplacian matrix that we will consider is weighted by a layout of the graph: the positions of the vertices in k-dimensional space. We denote a k-dimensional layout by a $n \times k$ matrix X. The position of the ith vertex is $X_i \in \mathbb{R}^k$. Lastly define a function $\mathbf{inv}(x) = 1/x$ when $x \neq 0$ and 0 otherwise.

Definition 2 *Let G be a graph and let X be a layout for G. The weighted Laplacian matrix L^X is an $n \times n$ matrix given by:*

$$L^X_{i,j} = \begin{cases} -w_{ij} d_{ij} \mathbf{inv}(\|X_i - X_j\|) & \text{if } i \neq j \\ -\sum_{k \neq i} L^X_{i,k} & \text{if } i = j \end{cases}$$

The off diagonal elements of L^X are $-w_{ij} d_{ij} \mathbf{inv}(\|X_i - X_j\|)$ where $\|X_i - X_j\|$ is the Euclidean distance between vertices i and j in the layout. The diagonal element $L^X_{i,i}$ is the positive sum of the off diagonal elements in row i.

The optimal layout corresponds to the global minimum of the following stress function.

$$\text{stress}(x) = \sum_{i<j} w_{ij} (\|X_i - X_j\| - d_{ij})^2 \tag{5}$$

The minimum of the stress is given by an iterative system involving weighted Laplacian matrices. Here Z is the current layout and X is the next layout. At the end of each iteration X is assigned to Z to compute a new X. The process continues until the stress function converges. A random layout can be used as the initial layout Z, but tends to require more iterations until convergence and is more likely to converge to a local minima.

$$L^w X = L^Z Z \tag{6}$$

Laplacian matrices are known to be positive-semidefinite with a one-dimensional null space spanned by $\mathbf{1} = (1, 1, \ldots, 1) \in \mathbb{R}^n$. Intuitively, the null space tells us that our stress function is invariant under

translation. To remove this degree of freedom, we remove the first row and column and consider only the $(n-1)\times(n-1)$ submatrix of each Laplacian matrix. This method requires us to set $X_0 = 0$ in the layout. As these submatrices are strictly diagonally dominant and positive-definite, we employ the conjugate gradient method to solve the system [9]. The conjugate gradient method is a popular algorithm for solving systems of linear equations of the form $Ax = b$, where x is an unknown vector, b is a known vector, and A is a known, square, symmetric, and positive-definite matrix.

Stress majorization produces layouts that approximate the actual graph metric because it considers the graph-theoretic distance between every two vertices. This method can easily account for weighted edges or other desired graph metrics, providing a clear advantage over the previously described spring-electrical model.

The requirement of access to the all-pairs shortest path matrix, d, severely constrains the scalability of stress majorization. Execution of Dijkstra's algorithm at each vertex allows computation of the APSP matrix in $O(VE + V^2 \log(V))$ time. Additionally, at each iteration we must compute L^Z, perform a matrix multiplication with Z, and use the conjugate gradient method to solve the system. As a result, stress majorization is not scalable beyond approximately 10^4 vertices. Low-rank Laplacian matrices have recently been used to extend stress majorization to much larger graphs [8]. An upcoming version of Graphviz will include an implementation of this extension.

Stress majorization is implemented in the Graphviz program *neato*. *neato* is suitable for weighted, or unweighted, undirected graphs.

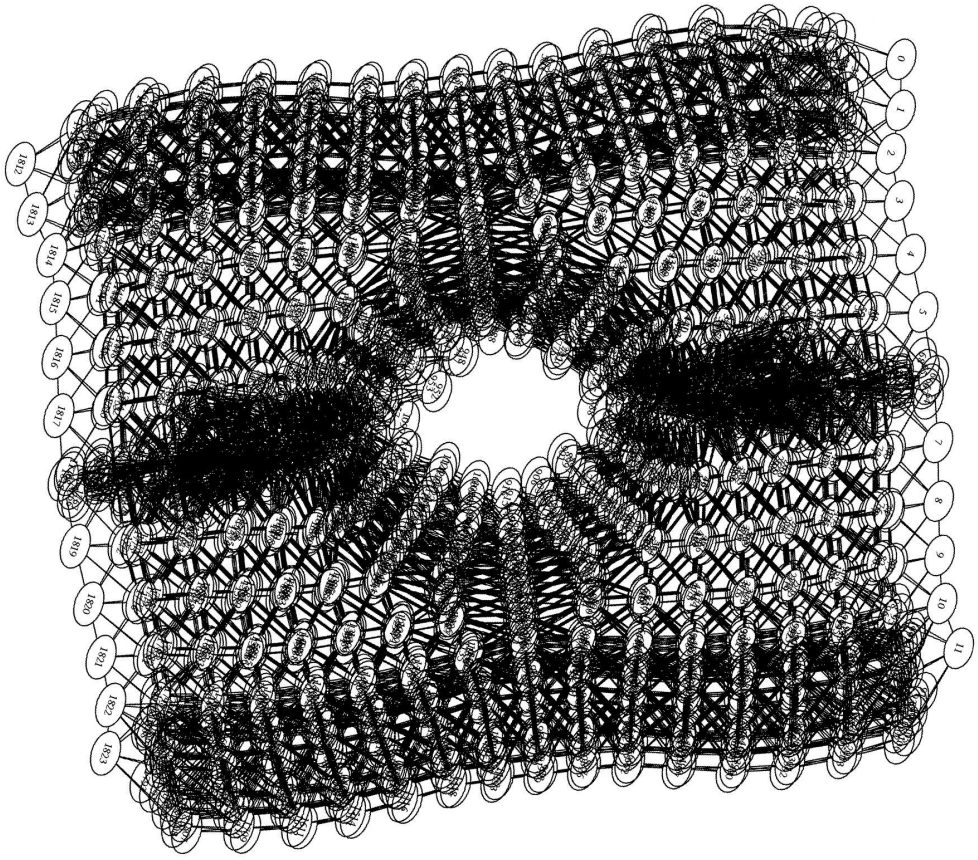


Figure 8: A drawing by *neato* of the nasa1824 dataset.

5 Reducing Edge Crossings

Thus far, we have limited our discussion to graph drawing algorithms for undirected graphs. Drawing directed graphs is an entirely new problem. The naive approach is to simply ignore the direction of the edges and use an algorithm for drawing undirected graphs. This is fundamentally unsatisfying because the directionality is frequently very important to effectively visualizing the graph.

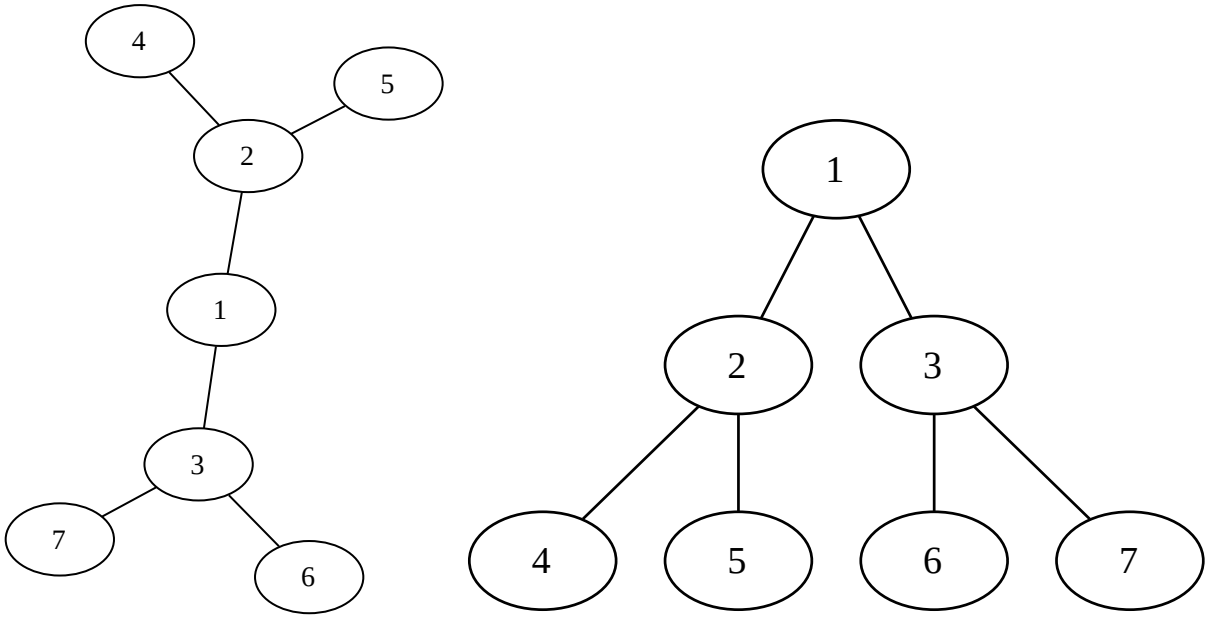


Figure 9: A comparison of the same graph drawn by *neato* (left) and by *dot* (right).

The Graphviz program *dot* uses a four-pass [6] algorithm for drawing directed graphs. *dot* requires that directed graphs be acyclic, and thus begins its layout algorithm by internally reversing edges that participate in many cycles. Note that this reversal is purely an algorithmic one: in the final drawing, the original edge direction is preserved. Ranks are then assigned to each vertex in the graph, and used to generate vertical coordinates in a top-to-bottom drawing. Ordering the vertices within each rank reduces the number of edge crossings, so heuristic methods are employed in order to find a good ordering. The horizontal coordinates are then assigned with the goal of keeping the edges short. Finally, splines are created for each edge.

The ranking process attempts to assign an integer rank $\lambda(v)$ such that for every $e \in E$ we have $l(e) \geq \delta(e)$ where the length $l(e)$ of $e = (v, u)$ is defined as $\lambda(u) - \lambda(v)$ and $\delta(e)$ represents a predefined minimum length constraint. Usually $\delta(e)$ is 1, but it may be any non-negative integer and can be set either internally or by the user.

A consistent ordering of vertices is possible only if the graph is acyclic. Since this is not guaranteed by the input, a preprocessing step must be executed in order to break any cycles in the graph. Depth-first search classifies every edge of a directed graph as either a tree edge, forward edge , cross edge, or back edge. It can be shown that there is a cycle in a graph if and only if there exists a back edge [4]. Heuristic methods can be used to reverse the direction of the most offensive back edges (those that participate in the most cylces).

We will define an optimal ranking as one where the edges are as short as possible. This ranking assignment may be formulated as an integer linear programming (ILP) problem.

$$\min \sum_{(v,u)\in E} w(v,u)(\lambda(u)-\lambda(v)) \tag{7}$$
$$constraint : \lambda(u)-\lambda(v) \geq \delta(v,u), \forall (v,u) \in E$$

To solve this ILP, Graphviz employs a network simplex formulation [3].

Having obtained a ranking for the vertices of the graph, we turn to the ordering of vertices within each rank. The ordering of these vertices determines the number of edge crossings in the drawing, so our goal is to compute an ordering with the minimum number of crossings. Unfortunately, solving this problem exactly is NP-complete, so heuristic methods are employed.

Computation of a good vertex ordering is an iterative process, with initial ordering computed by depth or breadth-first search starting at the vertices of minimum rank. Vertices are assigned positions within their ranks in left-to-right order as the search progresses.

Algorithm 2 Ordering()

```
order ← InitOrdering()
best ← order
for i = 0 to maxiter do
  WMedian(order,i)
  Transpose(order)
  if crossings(order) < crossings(best) then
    best ← order
  end if
end for
return best
```

The weighted median heuristic assigns each vertex a median based on the positions of the adjacent vertices in the previous rank. The median value of a vertex is defined as the median position of its adjacent vertices if that value is uniquely defined. If such a value is not defined, the median is interpolated. The weighted median heuristic biases the median value to the side where vertices are more closely packed. Finally, the vertices are sorted within their rank by their assigned median values.

Algorithm 3 WMedian(order,iter)

```
if iter is even then
  for r = 1 to maxrank do
    for v ∈ order[r] do
      median[v] ← MedianValue(v,r − 1)
    end for
    sort(order[r],median)
  end for
end if
```

The transpose operation iterates through each rank examining neighboring vertices within a rank. Two vertices are swapped if swapping them decreases the number of edge crossings in the layout. The transpose operation terminates when no swap operation will further decrease the number of crossings. Since a minimum must exist, termination is guaranteed.

Once a good ordering is found within each rank, all that remains is to assign x and y coordinates to each vertex and create the splines to draw the edges. The y coordinates are based on the rank, so vertices with the

Algorithm 4 Transpose(rank)

```
improved ← true
while improved do
  improved ← false
  for r = 0 to maxrank do
    for i = 0 to |rank[r]| -2 do
      v ← rank[r][i]
      w ← rank[r][i + 1]
      if crossings(v,w) > crossings(w,v) then
        improved ← true
        Swap(rank[r][i],rank[r][i + 1])
      end if
    end for
  end for
end while
```

same rank will appear on the same level. The x coordinates are determined using the ordering and chosen to keep the edges short. Finally, splines are computed for each edge.

6 Conclusion

Graphviz is an extremely powerful and widely used tool for visualizing graphs. Several algorithms have been presented above for drawing the most common types of graphs.

For small unweighted undirected graphs, *fdp* or *neato* will produce a good drawing. Once these graphs exceed 10000 vertices *sfdp* should be used. Unfortunately, *sfdp* does not take edge weights into account. To draw large weighted undirected graphs, low-rank stress majorization must be employed. Finally, *dot* is the tool of choice for visualization of directed graphs and trees.

7 Acknowledgements

I would like to thank my reviewer, Michael Schoenberg, whose helpful comments drastically improved this manuscript.

A Laplacian Matrices

The Laplacian matrix of a graph G is a representation of G similar to the adjacency matrix. In fact, the Laplacian matrix can be defined in terms of the degree matrix and the adjacency matrix. As we proceed, consider a simple graph G with $n = |V|$ and $m = |E|$.

Definition 3 *The adjacency matrix A for a graph G is the $n \times n$ matrix given by:*

$$A_{i,j} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

Definition 4 *The Laplacian matrix L for an unweighted, undirected graph G is the $n \times n$ matrix given by:*

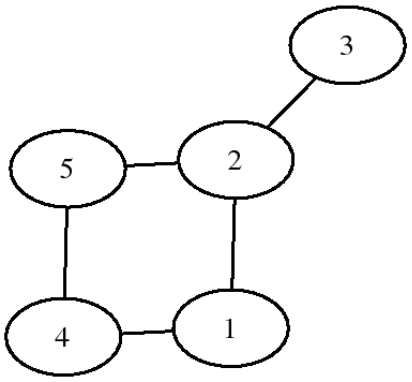
$$L_{i,j} = \begin{cases} \deg(i) & \text{if } i = j \\ -1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

where $\deg(i)$ is the degree of the ith vertex.

For each edge $(i, j) \in E$ the entry $L_{i,j} = -1$. Additionally, the degrees of each vertex are stored on the diagonal elements. The Laplacian matrix can also be expressed as

$$L = D - A$$

where D is the degree matrix and A is the adjacency matrix.



$$\begin{pmatrix} 2 & -1 & 0 & -1 & 0 \\ -1 & 3 & -1 & 0 & -1 \\ 0 & -1 & 1 & 0 & 0 \\ -1 & 0 & 0 & 2 & -1 \\ 0 & -1 & 0 & -1 & 2 \end{pmatrix}$$

Figure 10: A simple graph and its Laplacian matrix

All Laplacian matrices are positive-semidefinite and, as a result, have all nonnegative eigenvalues, $\forall_i \lambda_i \geq 0$. The number of times that 0 appears as an eigenvalue is equal to the number of connected components in the graph. Thus $\lambda_0 = 0$ for any Laplacian matrix. The first eigenvector v_0, corresponding to the eigenvalue λ_0, is always equal to $\mathbf{1} = (1, 1, \ldots, 1) \in \mathbb{R}^n$. Notice that $Lv_0 = 0$ because the sum of the elements in any row is 0. The null space of a Laplacian matrix is always nontrivial, but if G is a connected graph then the null space is a 1D subspace spanned by the vector $\mathbf{1}$. In general, the dimension of the null space is equal to the number of connected components in G.

References

[1] AT&T Labs. `http://www.graphviz.org/Documentation.php`.

[2] J. Barnes and P. Hut. A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324(6096):446–449, Dec. 1986.

[3] V. Chvatal. *Linear Programming*. W. H. Freeman, 1st edition, 1983.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[5] E. Gansner, Y. Koren, and S. North. Graph drawing by stress majorization. In J. Pach, editor, *Graph Drawing*, volume 3383 of *Lecture Notes in Computer Science*, pages 239–250. Springer Berlin / Heidelberg, 2005.

[6] E. R. Gansner, E. Koutsofios, S. C. North, and K. phong Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, 1993.

[7] Y. F. Hu. Efficient and high quality force-directed graph drawing. *The Mathematica Journal*, 10:37–71, 2005.

[8] M. Khoury, Y. Hu, S. Krishnan, and C. Scheidegger. Drawing large graphs by low-rank stress majorization. *Computer Graphics Forum*, To Appear.

[9] J. R. Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, 1994.