A Practical Approach to Static Node Positioning *

Junhui Luo and Kanth Miriyala

Center for Strategic Technology Research, Andersen Consulting
100 S. Wacker, Chicago, IL 60606, USA

Abstract. This paper discusses how we adapted an algorithm by Kernighan and Lin [4] and its refinement by Fiduccia and Mattheyses [1] for network partitioning to get an efficient heuristic for aesthetically and statically laying out undirected graphs.

1 Introduction

CASE tools generate and display information in the form of graphs, e.g., ER- and data flow diagrams, during interactive software evolution. To enable the user to quickly and easily understand the information presented, these graphs should be drawn aesthetically. Several layout styles and aesthetic criteria exist for drawing graphs. An extensive survey of this topic can be found in [6].

In this paper we deal with the static layout of undirected graphs. In static graph layout, a combinatorial description of the entire graph is read in and processed to produce, in one step, a layout for the entire graph. One approach to static layout of undirected graphs is the *spring embedder* [5, 7, 9]. The goal is to find a compact layout for straight line drawing style. Another approach is to use planarization algorithms combined with techniques for drawing planar graphs [11, 12]. The goal of planarization algorithms is to minimize the number of crossings between edges.

In this paper, we present a different approach from the above to satisfy a different set of aesthetic criteria:

- Non-overlapping of graph nodes.
- Even distribution of nodes over a region (a rectangular area) of given size.
- Minimization of the total distance between connected nodes in a given region.

We have chosen these criteria because the primary application areas we want to apply the layout technique to are, for example, ER- and data flow diagrams in the software engineering domain. In these applications meaningful clustering, compactness, and readability [8] are important characteristics of good graph layout. Also, these diagrams have been traditionally drawn using the *orthogonal graphic standard* [6], i.e., every node is a rectangle and every edge is comprised of horizontal and vertical segments. We have adapted an algorithm by Kernighan

* This work is supported in part by US Air Force, Rome Laboratory under KBSA/ADM program contract # F30602-93-C-0015.

and Lin [4] and its refinement by Fiduccia and Mattheyses [1] for network partitioning to get an efficient heuristic for aesthetically and statically laying out undirected graphs.

In Section 2 we state the input and output of the algorithm. In Section 3 we describe the static layout algorithm. In Section 4 we analyze the performance of our implementation, show some example layout generated by the system. Section 5 summarizes our contributions.

2 Problem Statement

The input undirected graph G to the layout algorithm is specified by its node set V_G and edge set E_G. Nodes of G have rectangular shape and have the same size.[2] The tasks of the layout algorithm are:

1. To determine the size of an appropriate rectangular area, R, henceforth called a region, in which G is to be laid out.
2. To determine the positions of all the nodes of G in the region R, subject to the primary aesthetic criteria stated in the previous section.

3 The Layout Algorithm

We first present an algorithm for laying out a graph G in a given region R in Sections 3.1 and 3.2. Discussion of region size estimation is deferred to Section 3.3.

3.1 Laying Out Graphs with More Than Four Nodes

We take a divide-and-conquer approach. The basic idea is to partition the graph and region into subgraphs and subregions respectively, assign a subgraph to a subregion. We recursively apply these steps to each subgraph and each subregion until there is only one node in a subgraph (and thus in a subregion). The center of the subregion is then considered to be the position of the node.

We adapted the Fiduccia and Mattheyses [1] algorithm to partition nodes in G into two blocks of the same size if the number of nodes in G is even, or with a difference of one if the number is odd. The algorithm is a fast approximation algorithm for the minimum cut partitioning problem[3] [10] so the number of edges connecting nodes in the two blocks is close to minimum. Details of the algorithm can be found in [4] and [1].

Splitting a region, R, into two subregions is straightforward. Regions are split horizontally and then vertically in an alternating fashion into two subregions of the same size. The decision to begin with a horizontal split is arbitrary. The strategy for assigning nodes to subregions is more complex. We would like to

[2] In Section 5 we discuss the consequences of having nodes of different sizes.

[3] The minimum cut problem is NP-Complete.

minimize the sum of distances between any one node in a subregion and another node which is connected to the first node but not in the same subregion. Because the exact positions of the nodes are unknown yet, we can only heuristically minimize the total distance between such node pairs.

Figure 1 illustrates the heuristic. Assume that we have already assigned node

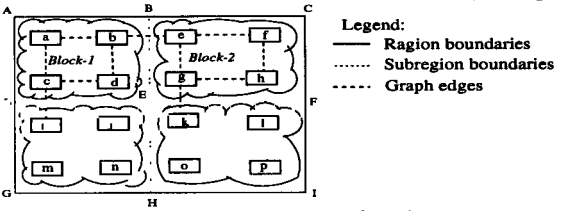

Fig. 1. Assignment of node blocks to subregions

set $\{i, j, m, n\}$ to subregion DEHG and $\{k, l, o, p\}$ to EFIH. Let S be the subgraph consisting of the node set, $V = \{a, \ldots, h\}$ and the edges $\langle x, y \rangle$ where both x and y belong to V. Assume that we have just partitioned V into *Block-1* consisting of a, b, c and d, and *Block-2* consisting of e, f, g and h. Also assume that we have split the region ACFD into subregions ABED and BCFE. The assignment of *Block-1* to subregion ABED and *Block-2* to BCFE brings c closer to i and g closer to k than the assignment of *Block-1* to BCFE and *Block-2* to ABED. Note that we use the center of DEHG to approximate the position of i and the center of EFIH to approximate the position of k. Since the node partitioning algorithm tends to place connected nodes in the same block, this heuristic is likely to reduce the total distance between connected nodes. In the above example we have assumed that $\{i, j, m, n\}$ and $\{k, l, o, p\}$ were assigned to their respective subregions before *Block-1* and *Block-2*. But if the order of processing was different, it is possible that we only knew that the node set $\{i, \ldots, p\}$ was assigned to DFIG. In that case, we use the center of DFIG to approximate the positions of i and k. The following outlines the algorithm.

Step 1. Partition nodes in V_G into two blocks $b1$ and $b2$ using the algorithm described in [1].

Step 2. Split region R horizontally (or vertically) into two subregions $r1$ and $r2$.

Step 3. Estimate (see remarks below) the total distance between nodes in R to connected nodes not in R when $b1$ is assigned to $r1$ and $b2$ is assigned to $r2$ by adding up estimated distances between each connected pair of nodes. Also estimate the total distance when $b1$ is assigned to $r2$ and $b2$ to $r1$. Choose the assignment with a lesser total distance.

Step 4. Recursively apply Steps 1 to 4 to each subgraph that contains more than four nodes.

A few comments are in order:

1. In the above algorithm, the exact location is not known for any node. Hence we have to use an approximate position for each node in Step 3. The approximate position of a node v is taken to be the center of the region that

v is currently assigned to, as explained in the example above. Note that as the blocks are split and assigned to smaller sized regions, this approximation becomes more accurate.

2. At the end of the above algorithm we have an assignment of blocks with less than or equal to 4 nodes to subregions. A consequence of splitting recursively until each block has less than or equal to four nodes is that the estimated distances for nodes become much closer to the actual distances. In the next section we will see that this degree of accuracy we achieved plays an important role in determining the final positions of nodes.

3.2 Laying Out A Subgraph of Four or Less Nodes

We motivate the need for a specialized heuristic for laying out graphs of four or less nodes with an example. Figure 2 shows the same graph drawn using different

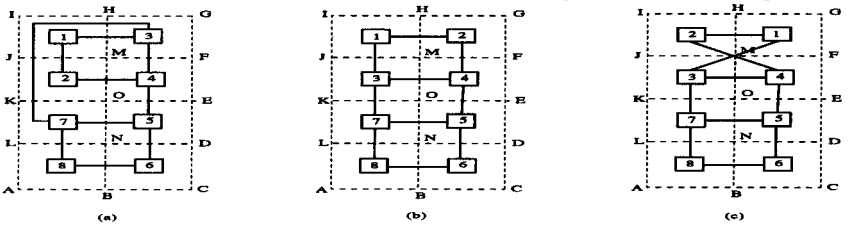

Fig. 2. Comparison of pure FM algorithm with special processing for four nodes

heuristics. Figure 2(a) has been drawn using only the Fiduccia and Mattheyses (FM) algorithm, i.e., if we had not terminated recursion in the algorithm described in the Section 3.1 when there were four or less nodes in a subregion but rather continued until there was only one node. Figure 2(b) has been drawn using the FM algorithm with a special heuristic for laying out subgraphs of four or less nodes. From the figure it is clear that the layout improves dramatically with the additional heuristic. In Figures 2(a), initially the nodes are split into two blocks $B1 = \{1, 2, 3, 4\}$ and $B2 = \{5, 6, 7, 8\}$. Then $B1$ is split into $B11 = \{1, 2\}$ and $B12 = \{3, 4\}$. The next successive partition splits $B12$ into $\{3\}$ and $\{4\}$. We assume that the four nodes in the bottom region, $\{5, 6, 7, 8\}$, have already been placed. Both 3 and 4 have external connections with respect to region $KEGI$. So regardless of whether $\{3, 4\}$ are placed in $KOHI$ or $OEGH$ there will be one long edge. Figure 2(a) shows the case where the two nodes are assigned to region $OEGH$.

Now we describe a heuristic for laying out a subgraph with four nodes which will achieve the layout illustrated in Figure 2(b). Assume that we are given a subgraph G_4 of the original graph G, to be laid out in subregion R. G_4 has four nodes. The basic idea is to partition R into four equal subregions at the horizontal and vertical medium and place one node in each subregion. As each of these subregions contains exactly one node we call these subregions *atomic*. They are not further split. We want to achieve two goals:

- If $v \in V_{G_4}$ is connected to $w \notin V_{G_4}$, we want to place v close to w. For example, in Figure 2, let $V_{G_4} be \{1, 2, 3, 4\}$. We want to place 3 close to 7 and 4 close to 5.
- If $u, v \in V_{G_4}$ are connected, we want to place them close to each other. In Figure 2(b), after nodes 3 and 4 have been positioned we want to position nodes 1 and 2. The layout in (b) is clearly better than the one in (c).

We define a number of functions and terms. For $v \in V_{G_4}$, $N_v = \{u|(u, v) \in E_G \& u \notin V_{G_4}\}$ is the set of all nodes not in G_4 which are connected to v. Node v is said to have *external connections* if N_v is not empty, otherwise v is said to be internal to G_4. *Region*(v) is the region currently assigned to v. Note that if the position of v has not yet been finalized, the center of *Region*(v) approximates the ultimate position of v. For two subregions r_1 and r_2, *center-distance*(r_1, r_2) is the distance between the centers of r_1 and r_2. If $v \in V_{G_4}$ and r is an atomic region of R, *primary-cost*$(v, r) = \sum_{u \in N_v}$ *center-distance*$(r$,*Region*$(u))$ gives an estimate of the total distance between v and all other nodes not in G_4 but are connected to v, if v is placed in r. An *assignment* is a particular way of placing nodes in V_{G_4} to the atomic regions. For a particular assignment, *total primary cost* is the sum of primary costs over the nodes having external connections. For a particular assignment, internal node v of G_4, and atomic region r of R, the *secondary cost* of placing v in r is the total distance from the center of r to the centers of all other atomic regions of R that contain a node which is connected to v. For a particular assignment, *total secondary cost* is the sum of secondary costs over all the internal nodes of G_4.

The algorithm outlined below achieves the above goals and determines the final positions of nodes in V_{G_4}.

Step 1. Split the input region horizontally and vertically at the middle into four atomic regions.

Step 2. For each $v \in V_{G_4}$ and each atomic region r of R, compute and store *primary-cost*(v, r) in a 4×4 array.

Step 3. Place each of the four nodes that has an external connection into a different atomic region in such a way as to minimize the total primary cost. This step is implemented by a branch-and-bound search algorithm which uses the *primary-cost* array.

Step 4. Place each internal node of G_4 in an atomic region of R not taken in Step 3 in such a way as to minimize the total secondary cost. This step is carried out by comparing all possible assignments and choosing the one which has the minimum total secondary cost.

We omit descriptions for laying out a subgraph with two or three nodes, as they are very similar to that for laying out four nodes. The difference is in the way a region is partitioned.

3.3 Region Estimation and Preprocessing

In the previous two subsections, we assumed that the region R in which to layout graph G was given. Now we describe how the size of R is determined.

Given G, width and height of the region R are computed based on the number of nodes, N, in G and the width w and the height h of each graph node: $width = c \times w \times n$, $height = c \times h \times n$ where $n = 2^{\lceil (\log_2 N)/2 \rceil}$, and constant c is empirically chosen to leave enough space between nodes, e.g., if $c = 1.5$, then the space between two nodes positioned horizontally next to each other is half the width of a node and the space between two nodes positioned vertically to each other is half the height of a node. Note that $2^{\lceil (\log_2 N)/2 \rceil}$ is the maximum number of nodes that can be aligned horizontally or vertically by our layout algorithm. Therefore the size of R so estimated will guarantee non-overlapping of graph nodes if c is greater than 1.

If the assumption that all nodes are of the same size does not hold, then we can use the maximum node width and height in the above computation to guarantee non-overlapping. The problem with this is, however, that if the variation in node size is significant, inter-node distance may vary accordingly and nodes may not look evenly distributed over the layout region. Currently we use a technique called *space creation and compaction* [3] to remedy this anomaly.

This concludes our description of the layout algorithm. We would like to mention that when an input graph is not a connected graph, it is desirable that each connected component of the graph be laid out in a different region so that connected components can be easily identified. Unfortunately, the algorithm described so far does not have this property. Nevertheless, this can be achieved by first identifying connected components of the input graph and then laying out each component in a different region.

4 Performance and Analysis of the Algorithm

We have implemented the above algorithm on Sun Sparc Stations in C++. It takes the implementation less than one second to compute node positions of randomly generated graphs comprised of hundreds of nodes and edges. In fact, the time complexity of the layout algorithm is $C(n + e) \log n$, where n is the number of nodes in the graph and e the number of edges, C is bounded by e but in practice is usually a small number, see [4]. Figure 3 shows the result of our

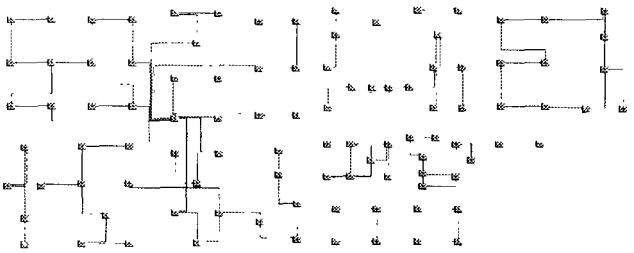

Fig. 3. Example result of layout

algorithm when applied to an example graph[4].

[4] Edges are routed by the edge routing algorithm described in [2].

To assess the effect of the special processing described in Section 3.2 of subgraphs with at most four nodes, we applied two versions of the node positioning algorithm to randomly generated graphs, one includes the special processing and the other does not, but rather applies the Fiduccia and Mattheyses algorithm to partition graphs of all sizes. We gathered the following statistics.

- Sum of distances between connected nodes.
- Sum of actual lengths of all edges routed by the edge-routing algorithm.
- Total number of edge bends.
- Total number of edge crossings.

Note that only the first item on the list is the measurement that the node positioning algorithm directly tries to minimize. The other measurement items contribute to overall aesthetic graph layout but are not under direct control of the node positioning algorithm. They are determined by the edge routing algorithm[2]. Nonetheless, from the statistics we can verify the degree of interplay between node positioning and edge routing. Table 1 gives the size of 15 randomly generated graphs and Figure 4 graphically shows the results we have collected. From the charts, we can see that the total distance between connected nodes

Table 1. Graph sizes

Graph	G1	G2	G3	G4	G5	G6	G7	G8	G9	G10	G11	G12	G13	G14	G15
# of Nodes	40	40	40	60	60	60	80	80	80	100	100	100	120	120	120
# of Edges	61	71	81	93	106	120	123	141	161	156	177	201	186	213	241

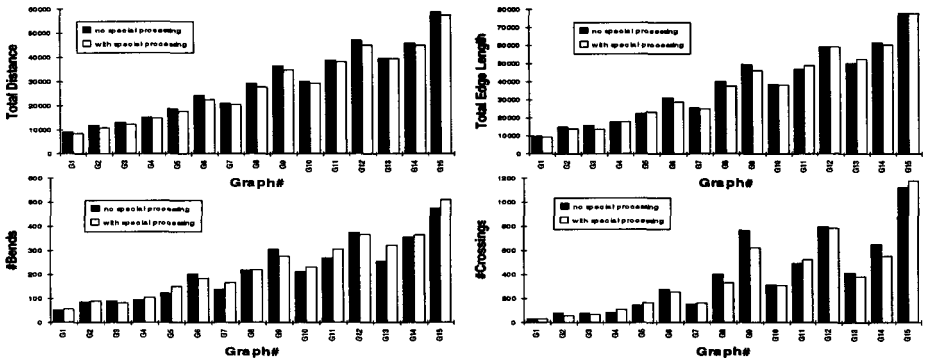

Fig. 4. Comparison of pure FM algorithm and special processing algorithm

uniformly decreased when we switched from without special processing to with special processing, affirming the effectiveness of the special processing. On the other hand, the total edge length did not uniformly decrease - Four cases (G4, G5, G11, G13) show an increase. This indicates that closeness of connected nodes does not necessarily imply shorter edge length.

Only four (G3, G6, G9, and G12) cases showed decrease in number of bends when we switched from without special processing to with special processing. Nine (G2, G3, G6, G8, G9, G10, G12, G13, G14) cases showed decrease in

number of crossings. This indicates that overall the special processing did not improve these aesthetic criteria due to some global edge interactions, although locally, as we have explained in Sections 3.2, these measurements should improve.

5 Conclusions and Acknowledgement

We described an efficient algorithm for automatically positioning graph nodes in an undirected graph. It has been incorporated in a number of applications.

We thank Roberto Tamassia and Scot Hornick for their contributions to defining the problem, Michael DeBellis and Edy Liongosari for supporting and managing the project, Gadi Friedman for implementing a prototype of the algorithm in Lisp, Bhaskar Naidu and Jung Kim for various discussions, suggestions and help during the course of this project. We also thank Subrata Mitra and Amitabh Dave for their valuable comments on an earlier draft of this paper.

References

1. Fiduccia,C.M. and Mattheyses,R.M., *A Linear-Time Heuristic for Improving Network Partitions*, Proceedings of the 19th Design Automation Conference, IEEE, 1982, pp.175-181.
2. Miriyala,K. Hornick,S.W., and Tamassia,R, *An Incremental Approach to Aesthetic Graph Layout*, Proc. of 6th Intl. Workshop on Computer-Aided Software Engineering, 1993, pp.297-308.
3. Naidu,B. and Miriyala,K., *Space Creation and Compaction in Dynamic Diagramming*, Andersen Consulting Internal Paper.
4. Kernighan,B.W. and Lin,S.: *An Efficient Heuristic Procedure for Partitioning graphs*, Bell System Technical Journal, Vol. 49, Feb. 1970, pp.291-307.
5. Eades,P.: *A Heuristic for Graph Drawing*, Congressus Numerantium, Vol. 42, pp. 149-160, 1984.
6. Di Battista, Eades,P., and Tamassia,R.: *Algorithms for Drawing Graphs: an Annotated bibliography*, unpublished paper, March, 1993.
7. Fruchterman,T.M.J. and Reingold,E.M.: *Graph Drawing by Force-Directed Placement*, Software Practice and Experience, Vol. 21, No. 11, pp. 1129-1164, 1991,
8. Tamassia,R., Di Battista,G., and Batini,C.: *Automatic Graph Drawing and Readability of Diagrams*, IEEE Trans. on Systems, Man, and Cybernetics, Vol. 18, No. 1, pp. 61-79, 1988.
9. Kamada,T.: *On Visualization of Abstract Objects and Relations*, Ph.D. thesis, Department of Information Science, University of Tokyo, 1988.
10. Garey,M., and Johnson,D.: *Computers and Intractability: a guide to the theory of NP-completeness*, W. Freeman and Company, 1979.
11. Cai,J.,Han,X., and Tarjan,R.E.,: *An O(m log n)-time Algorithm for the Maximal Subgraph Problem*, SIAM J. on Computing, 1993.
12. Batini,C., Nardelli,E., Talamo,M., and Tamassia,R.: *A Graphtheoretic Approach to Aesthetic Layout of Information Systems Diagrams*, Proc. 10th Intl. Workshop on Graphtheoretic Concepts in Computer Science (Berlin, June 1984), pp.9-18, Trauner Verlag, 1984.