THE FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

CROSSING REDUCTION FOR LAYERED HIERARCHICAL GRAPH DRAWING

By

PITCH PATARASUK

A Thesis submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Master of Science

Degree Awarded:
Summer Semester, 2004

The members of the Committee approve the Thesis of Pitch Patarasuk defended on April 29, 2004.

Daniel Schwartz
Professor Directing Thesis

Lois Hawkes
Committee Member

David Whalley
Committee Member

The Office of Graduate Studies has verified and approved the above named committee members.

ACKNOWLEDGEMENTS

I would like to thank Dr. Daniel Schwartz for being my major professor and all his help and support that enabled me to complete this thesis. I would like to thank Dr. David Whalley and Dr. Lois Hawkes for their kindness in serving on my committee.

Thanks to the Computer Science Department, FSU library, and FSU computer lab for all their conveniences, which helped me achieve my goal.

Thanks to my family and friends for all of their support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Graph drawing is widely used in many fields. Good visualization in graph drawing makes it easier for humans to understand the concepts being represented. This thesis deals with what is known as a hierarchical graph and concerns one of the core issues in making graphs look better, that of reducing the number of edge crossings.

The general problem of crossing reduction is discussed, and several approaches are implemented and compared. The overall approach uses the well-known "layer-by-layer sweep" algorithm, where layers are generally considered two at a time and some heuristic is applied for reducing the numbers of edge crossings between just those two layers.

Two specific one-sided 2-level crossing reduction heuristics were implemented: the permutation heuristic and the barycenter heuristic. Both heuristics were each implemented in two ways, with and without randomness. Tests were run which demonstrate that introducing randomness in this manner significantly improves the performance of both heuristics. Moreover, the randomized barycenter method is preferable due to its ability to deal efficiently with large graphs.

Then an alternative, hybrid approach was implemented. This uses the same layer-by-layer-sweep but applies the permutation heuristic when the number of vertices in the given layer is small, and applies the barycenter heuristic otherwise. In general, it did not show significant improvement over the randomized barycenter heuristic. Finally, a new alternative is proposed for future research.

INTRODUCTION

Overview

Graph drawing is widely used in many fields. Good visualization in graph drawing makes it easier for humans to understand the concepts being represented. This thesis deals with what are known as a hierarchical graph and concerns one of the core issues in making graphs look better, that of reducing the number of edge crossings.

Some definitions are reviewed and the approach of creating layered graph drawings is discussed. Next is considered the general problem of crossing reduction. Obtaining an optimal result for an arbitrary hierarchical graph is known to be computationally too expensive. A standard alternative is to employ a “layer-by-layer sweep”, where layers are generally considered two at a time and some heuristic is applied for reducing the numbers of edge crossings between just those two layers. The layer-by-layer-sweep process is performed first top-to-bottom then bottom-to-top, and this is iterated until some termination condition is satisfied.

The general layer-by-layer sweep algorithm was implemented in such a way that it could be employed with any one-sided 2-level crossing reduction heuristic where one layer is fixed and the other is re-ordered. Then two specific heuristics were implemented: the permutation heuristic and the barycenter heuristic.

The permutation heuristic considers all possible permutations of the layer to be re-ordered. This produces an optimal result but has the drawback that it is NP-complete. It can be applied effectively only for graphs that are narrow. The barycenter heuristic re-orders vertices according to the average values of their neighbor’s order. It is the most popular method because it is efficient and easy to implement and, in most cases, produces good results. A drawback, however, is that it can have a worst case output where the number of edge crossings is as many as $\Omega(\sqrt{n})$ times the optimum number of crossings, where n is a number of all vertices in an entire graph.

These two heuristics were each implemented in two ways. In the permutation heuristic, there can be more than one ordering of the vertices that produce the best crossing reduction. One approach is to select the first one of these encountered. The other is to randomly select one from among the set of all candidates. For the barycenter heuristic, two vertices can vie for the same position in the final arrangement. One approach here is to list them in the original order in which they are found. The other is to pick a random rearrangement.

Tests were run which demonstrate that introducing randomness in this manner significantly improves the performance of both heuristics. Moreover, it was seen that the randomized barycenter method is preferable due to its ability to deal efficiently with large graphs.

Then an alternative, hybrid approach was suggested and implemented. This uses the same layer-by-layer-sweep algorithm but applies the permutation heuristic when the number of vertices in the given layer is small, and applies the barycenter heuristic otherwise. This was tested and compared with the former two. In general it did not show significant improvement over the randomized barycenter heuristic. However, it proved that a hybrid method was worth considering, especially if one could find an effective hybrid method. Then, a new alternative is proposed for future research.

All algorithms in this work were implemented in Java, with the displays being drawn using Java 2D.

Organization of the Thesis

Chapter 1 talks about graphs in general. Chapter 2 discusses how to produce a layered graph drawing. Chapter 3 covers the general notion of k-level crossing reduction. Chapter 4 contains an overview of existing heuristics for 2-level crossing reduction, and provides details of the permutation and the barycenter heuristics. Chapter 5 explains an implementation of the permutation heuristic and the barycenter heuristic. Chapter 6 presents the suggested new method for crossing reduction. Chapter 7 compares results from the existing and suggested algorithms. Further work is suggested in Chapter 8.

CHAPTER 1

LAYERED HIERARCHICAL GRAPH DRAWING

General Graph Drawing

There are many conventions for drawing graphs. Some widely used ones are (cf. [1]):

- Straight-line drawing: each edge is drawn in a straight line from one vertex to another
- Poly-line drawing: each edge is drawn as a polygonal chain
- Orthogonal drawing: each edge is drawn as a polygonal chain of alternating horizontal and vertical segments
- Grid drawing: vertices, crossing, and edge bends have integer coordinates
- Planar Drawing: no two edges cross

Some examples are shown in Figure 1.1 (also from [1]).

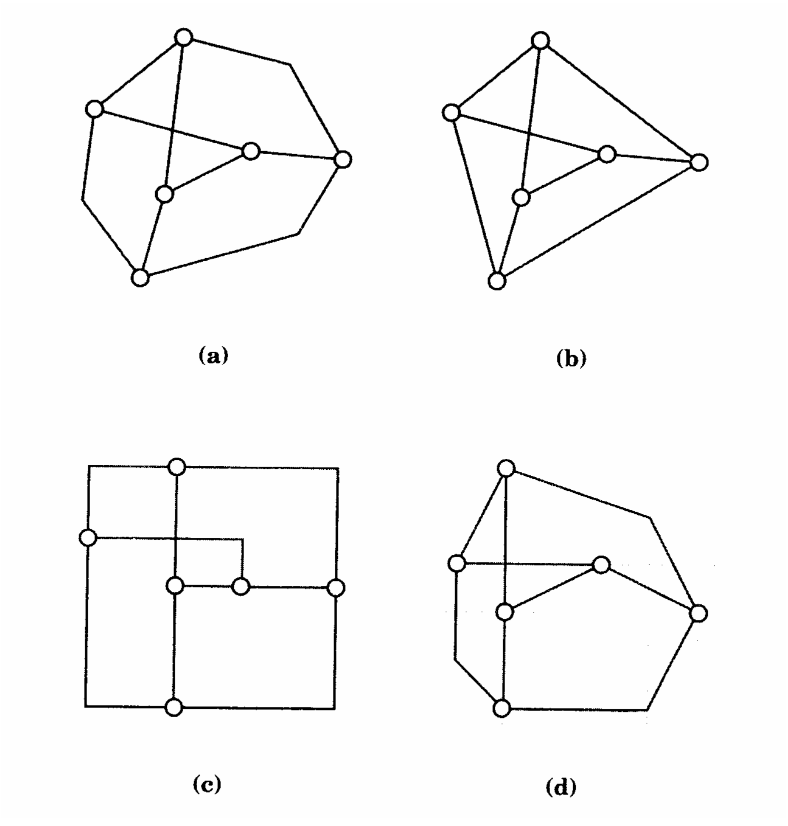

Figure 1.1: Drawings for the same graph: (a) poly-line, (b) straight-line, (c) orthogonal, and (d) grid

Certain issues that should be considered for drawing graphs are (from [3]):

- Display symmetry
- Avoidance of edge crossings
- Avoidance of bends in edges
- Keeping edge lengths uniform
- Distributing vertices uniformly

Planar Graph

A planar graph is a graph that can be drawn with no edge crossing. Planar graphs play an important role in graph drawing. In practice, if a graph can be drawn as a planar graph, it should be drawn as a planar graph. There are several algorithms to draw non-planar graphs that planarize the graphs first, and then apply a planar graph-drawing algorithm.

Directed Graph

A directed graph, or digraph, is one in which each edge has an arrow at one end. The arrow shows the direction of the edge.

Directed Acyclic Graph

A directed acyclic graph (DAG) is a directed graph in which no set of edges forms a loop.

Hierarchical Graph

Hierarchical graph is generally a DAG that displays hierarchy. Each edge in such graph does not display its arrow since the parent-child relationship implies the direction of each edge.

Layered Hierarchical Graph Drawing

Layering refers to a manner in which graphs may be drawn for visual display. Because a hierarchical graph organizes its vertices into a hierarchy, they can be drawn so that this hierarchy is displayed as a collection of layers. Since, in general, a child node can have any number of parents; this can lead to graphs with many crossing lines. Thus an objective of layered graph drawing is to minimize, or otherwise reduce, the number of edge crossings.

CHAPTER 2

AN APPROACH TO GENERATE LAYERED GRAPH DRAWING

An Approach to Generate Layered Graph Drawing

The layered drawing of a graph may be divided into three parts. This analysis is adapted from [1].

- Layer Assignment
- Crossing Reduction
- Horizontal Coordinate Assignment

Layer Assignment

This assigns each vertex with a y-coordinate. These coordinates are numbered 1 through k, where k is the number of layers of a graph. There are some requirements for the layering.

1. The layer should be compact. This means that the width of each layer should be small.

2. The layer should be proper. This means that there is no edge between any vertices that span more than one layer. This can be achieved by inserting dummy vertices. These dummy vertices will act as real vertices for a purpose of crossing reduction, but they will not be displayed in the final drawing. An example of a proper graph after dummy vertices have been inserted is shown in Figure 1.2.

3. The number of dummy vertices should be small. The reasons are:

    - The time used for crossing reduction depends on the number of vertices, which includes dummy vertices.
    - Dummy vertices will create bends in the final drawing. There is an algorithm to change bends into straight-lines, but this will affect the total time required to create the graph.
    - Dummy vertices make edge lines longer, which make it harder for human eyes to follow.

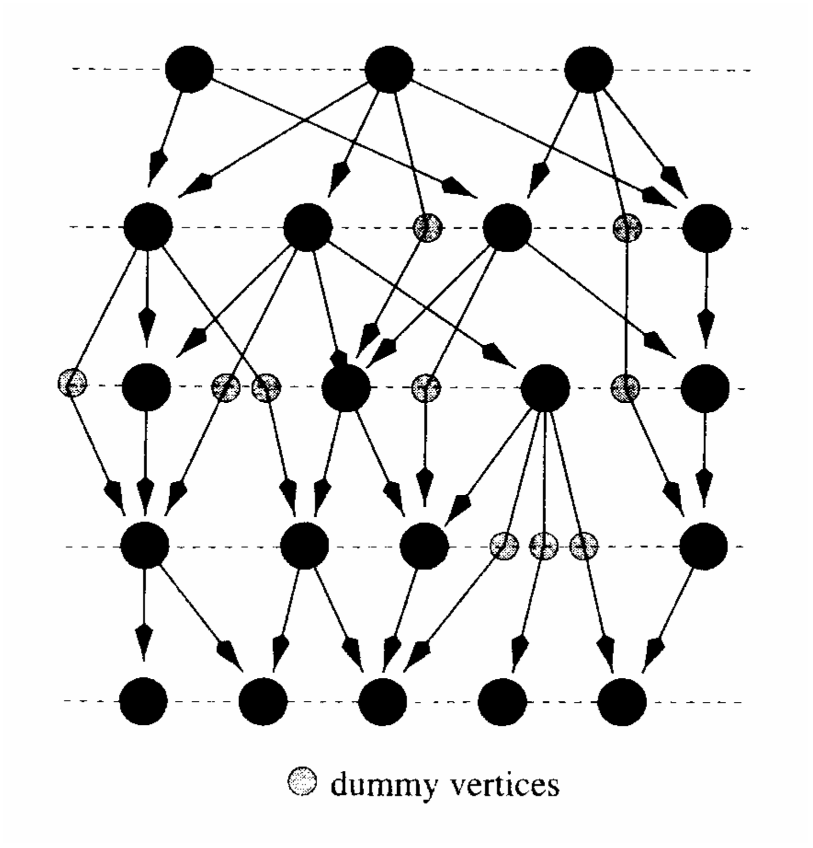

Figure 1.2: A proper graph

There are 3 techniques to compute layering.

- The longest path layering: Here the layer number of a vertex is given as the length of the longest path from that vertex to a root vertex plus 1 (root vertices are at layer 1). This method is simple to implement and will give the smallest number of layers. However, this method will also produce a graph with maximum width.
- Layering to minimize width: This method tries to minimize widths of all layers. It does this by adding dummy vertices to move certain sections of the graph downward, thus reducing the number of vertices at a given layer, but increasing the graph's overall height. Having a smaller number of vertices in various layers reduces the time required during the crossing reduction step. However, to find a layering of a graph with both minimized width and minimized height is NP-complete.
- Minimizing the number of dummy vertices: The reasons of minimizing number of dummy vertices are not only those described earlier, but also that this makes the graph more compact.

Hybrids methods of these techniques are also used.

Crossing Reduction

Crossing reduction has been considered as a fundamental issue for graph drawing. Fewer numbers of crossings give a better visualization of a graph. This step will be discussed in the following chapters.

Horizontal Coordinate Assignment

This step converts an ordering of each layer into x-coordinates. This step also tries to reduce bends in edges that are generated by dummy vertices.

CHAPTER 3

K-LEVEL CROSSING REDUCTION

Crossing Reduction

Hierarchical graphs with a small number of crossings are easier for readers to understand. Crossing reduction algorithms for an entire graph, which are called k-level crossing reduction algorithms, are practically based on algorithms for 2-level crossing reduction, where the latter reduces the number of crossings for two input layers. The "layer-by-layer-sweep" is a general format that applies 2-level crossing reduction techniques to reduce a number of edge crossings for an entire graph and will be discussed in this chapter.

The input to a crossing reduction algorithm has to be a proper graph, as described earlier. The number of crossings occurring in a graph depends on the ordering of the vertices in the layers. The problem of minimizing edge crossing in a layered graph is NP-complete, even if there are only two layers. Furthermore, it is NP-complete for a k-layered graph even if there is only one non-dummy vertex in each layer. There are several more-efficient algorithms that try to reduce the number of edge crossings without completely minimizing them. These algorithms will be discussed later.

Layer-by-Layer-Sweep

The most common technique for reducing the number of crossings in a k-layered graph is called the layer-by-layer-sweep. This method works as follows. Suppose we have k layers, with the topmost being layer 1 and the bottommost being layer k. First, we initialize an ordering of each layer by labeling the vertices from left to right as 1,2,…,n, where n is the number of vertices of that layer including dummy vertices. Then, for i = 2,…,k, we fix the layer L_{i-1} and use a one-sided 2-level crossing reduction method to find a new ordering of layer L_i. After this has been done for all layers, we start over at layer k and reverse the method, traversing back up through the layers, by fixing each layer L_i and finding a new ordering of layer L_{i-1}. This process continues, traversing down and up, until no better result is achieved.

Alternative Methods

There are many alternative methods of doing a crossing reduction for layered graph drawing. Di Battista et al. [1] suggested a method that considers three layers at a time by fixing a top and a bottom layer and trying to find a new ordering for the middle layer.

There are many studies of two-sided crossing reduction for 2-level graphs. Instead of having one layer fixed, these algorithms try to re-order both layers at the same time. More information can be found in a paper by Mutzel and Junger [5]. These algorithms can be applied to k-level crossing reduction by traversing upward and downward as in the layer-by-layer sweep method.

Matuszewski, Schonfeld, and Molitor [6] introduced the use of sifting for k-level crossing reduction. This method is normally used for dynamic ordering, where the user can input vertices and edges at run-time.

There are several crossing reduction algorithms based on planar graphs. Many algorithms planarize the graph by removing a few vertices or edges, so as to make the graph planar. Then, a planar graph crossing reduction algorithm is applied. After that, the removed vertices and edges are placed back into a final graph. More information can be found in [4, 7].

A Method that Produces the Best Non-optimal Result

There is no algorithm that can produce the best result in practical time. The only approach that guarantees to get a best result is to permute all vertices in all layers. The time for this will be bound by n-factorial, where n is a number of vertices in the entire graph. A practical approach that can produce a nearly optimal result was proposed by Mutzel and Junger [2]. This uses the layer-by-layer sweep algorithm along with the branch-and-cut heuristic. This method is considered to be the best among those that are not optimal. However, it is still impractical to use for very large graphs.

CHAPTER 4

2-LEVEL CROSSING REDUCTION

One-sided 2-Level Crossing Reduction Heuristics

As contended in the previous chapter, most algorithms for crossing reduction are based on algorithms for 2-level crossing reduction. One type of the latter is called one-sided 2-level crossing reduction, which fixes one layer and tries to re-order the other. Some existing heuristics for this are as follows, cf. [5, 6].

- Branch and cut heuristic: uses the ABACUS branch-and-cut system. It produces an optimal result for one-sided 2-level crossing reduction.
- Median heuristic: re-orders vertices according to the median values of their neighbors' order.
- Greedy-insert heuristic: chooses the next vertex to be inserted in such a way that it minimizes the number of crossing generated from edges adjacent to the vertex and all vertices to the left of that vertex.
- Greedy-switch heuristic: switches consecutive pairs of vertices if this would reduce the number of crossings.
- Split heuristic: chooses a pivot vertex and puts the rest of the vertices either to the left or right of that pivot vertex depending on which side would make a smaller number of crossings. This method continues recursively.
- Assignment heuristic: computes the entries of vertices based on an adjacency matrix and on a four dimensional matrix.
- Permutation heuristic: permutes all vertices and choose the ordering that generates the fewest number of crossings. It produces an optimal result for one-sided 2-level crossing reduction.
- Barycenter heuristic: re-orders vertices according to the average values of their neighbors' order.

This thesis concentrates on the last two algorithms.

Permutation Heuristic

This fixes one layer and permutes all vertices in another layer. It finds the number of crossings for each permutation. The output ordering is an ordering that has the fewest number of crossings. This method produces the best result for each two input layers with one layer fixed. However, this method is NP-complete. The time and space uses will be bound by n-factorial (n!), where n is a number of vertices in the layer being re-ordered. This is impractical for layers with a large number of vertices including dummy vertices.

As an example, if the layer to be re-ordered has three vertices, X, Y, and Z, these will be permuted in six different ways (3-factorial), XYZ, XZY, YXZ, YZX, ZXY, and ZYX.

An ordering that produces the fewest edge crossings will be the output ordering of this method.

Barycenter Heuristic

This heuristic re-orders vertices according to the average values of their neighbors' order. The example below fixes a top layer and tries to re-order a bottom layer. The initial ordering for the top layer are 1, 2, 3, and 4 for vertices A, B, C and D, respectively. An initial ordering for the bottom layer are 1, 2 and 3 for vertices X, Y, and Z. We now calculate barycenter values (average) of vertices X, Y, and Z. For vertex X, its parents have an order of 2 and 3; hence, the barycenter value, which is the average value of its parents' orders, is 2.5. By using the same method, the barycenter values for vertices Y, and Z are 3 and 1, respectively.

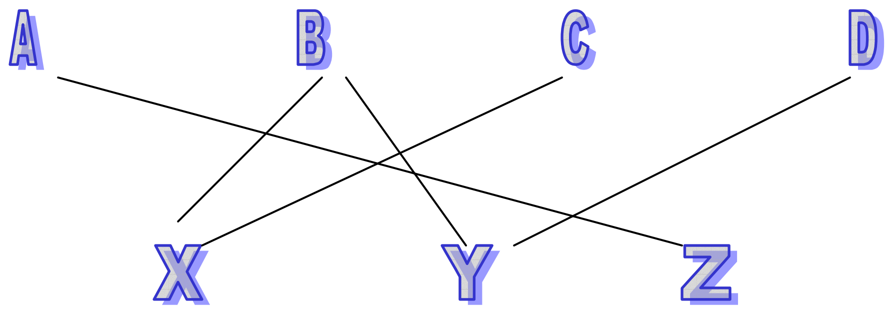

Figure 4.1: Barycenter heuristic

After we get barycenter values for all vertices in the bottom layer, we sort the vertices according to their barycenter values. The new ordering of the bottom layer is now Z, X, Y. This ordering reduces a number of edge crossings from 5 to 1.

It is a well-known fact that for one-sided 2-level crossing reduction, if a planar graph drawing is possible, the barycenter heuristic will produce a planar drawing.

The barycenter heuristic is considered the most popular one for one-sided 2-level crossing reduction because of its fine output and its fast computational time. According to [1], it can be computed in linear time. Inasmuch as this algorithm incorporates a sort routine, however, it seems that this should actually be n log n.

In theory, an output of the barycenter method can have the number of edge crossings be as many as $\Omega(\sqrt{n})$ times the optimum number of crossings [8]. However, this heuristic normally generates good results.

CHAPTER 5

AN IMPLEMENTATION OF PERMUTATION HEURISTIC AND BARYCENTER HEURISTIC

An Implementation of Permutation Heuristic

An Implementation of the Layer-by-Layer-Sweep

To study how crossing reduction works, it was decided to start by trying to implement the permutation heuristic. This required an implementation of a layer-by-layer-sweep algorithm. A few problems arose.

First there is no full explanation of how to implement a layer-by-layer-sweep. Most writings only describe this algorithm briefly, somewhat as follows. First the sweep starts at the top by traversing downward considering two layers at a time. After it reaches the bottom, it traverses back upward. The sweep continues until no further refinement can be achieved. The problem is that it is nowhere stated exactly what is meant by “no further refinements can be achieved”. Initially, this author believed that the sweep would produce a better result for each traversal until it reaches the best result it can find with the heuristic it uses, i.e., the number of crossings would be reduced by each sweep and terminate when some result was repeated. It turned out that this was a misconception.

Consider the graph in Figure 5.1 (modified from Reddy [9]). Throughout this thesis, this will be referred to as the “initial graph”.

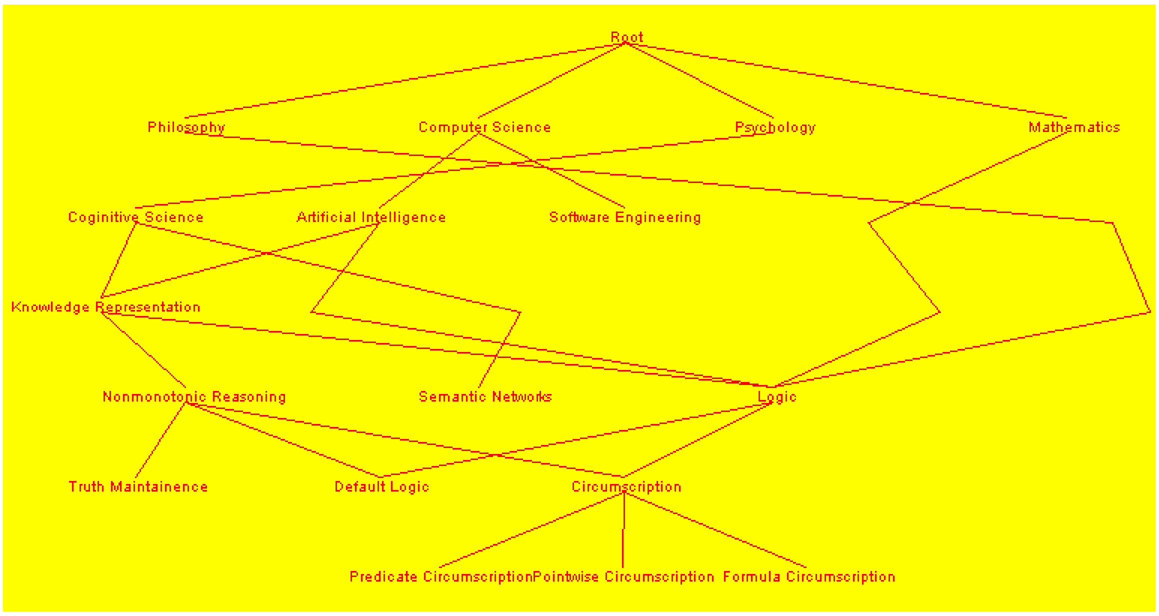

Figure 5.1: The initial graph

Figure 5.2 shows the result of starting with the initial graph and performing the first downward sweep of the layer-by-layer-sweep with permutation heuristic. This graph has three crossings. Figure 5.3 is the result of performing an upward sweep from the graph in Figure 5.2. This has six crossings. Thus, assumption that successive passes produce better results is incorrect.

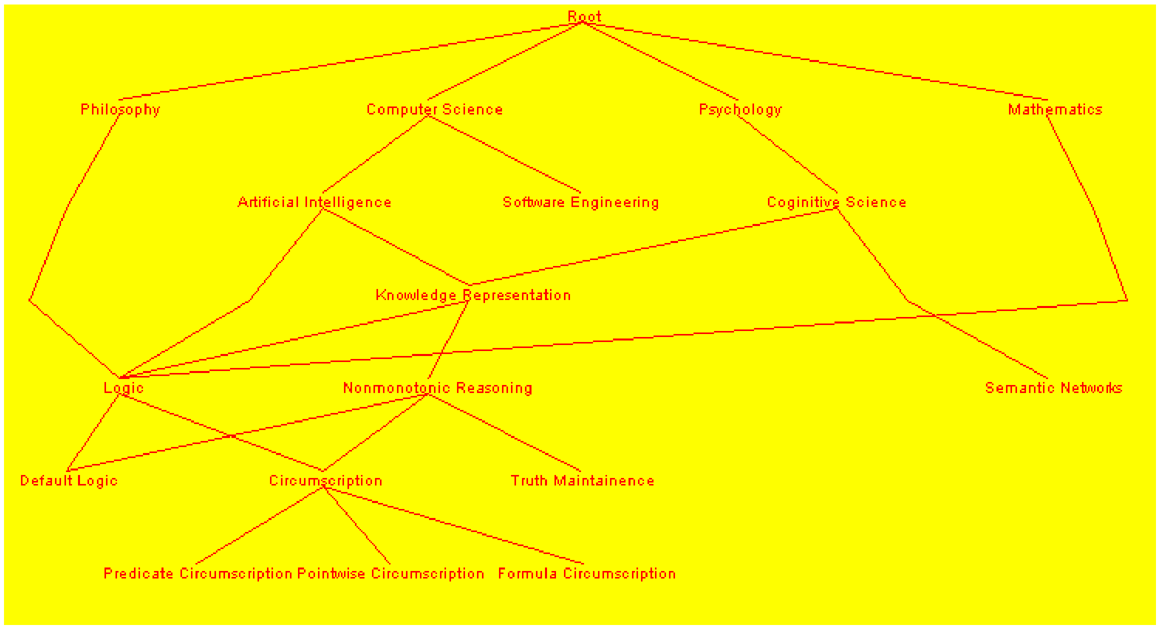

Figure 5.2: An output from a program that implements "permutation heuristic" with one sweep downward

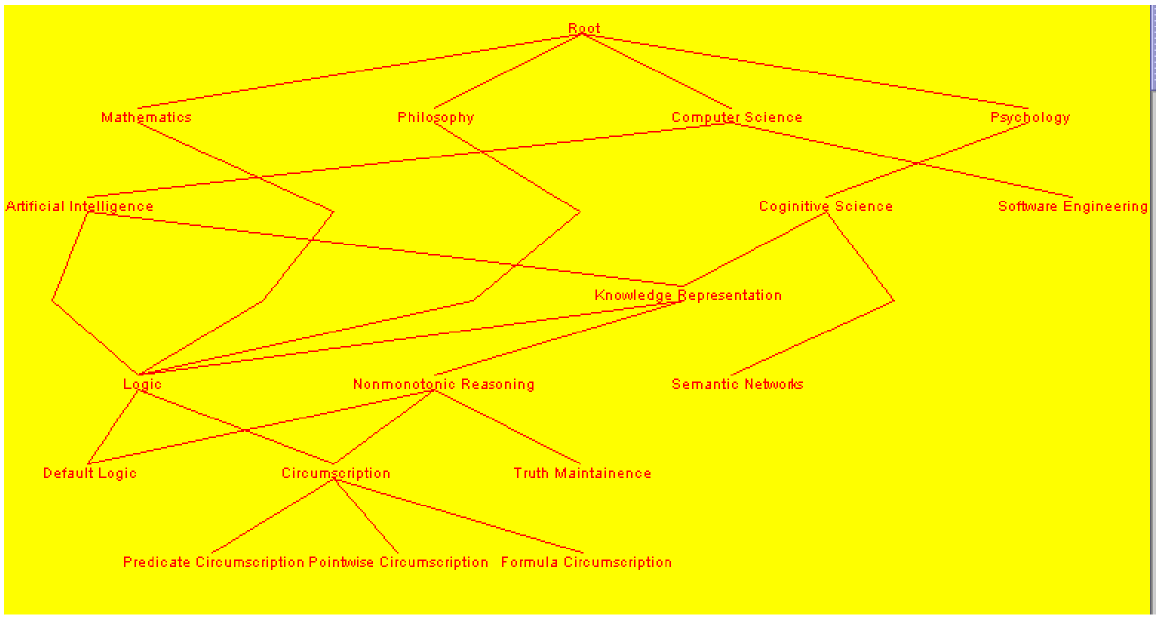

Figure 5.3: An output from a program that implements "permutation heuristic" with one sweep downward, then one sweep upward

Moreover, it turns out that, even after producing a worse result than on a previous sweep, further sweeps can actually find further improvements over all results found before. Indeed, in the present example, if one additional downward sweep is performed, then the result is as shown in figure 5.4, which has only one crossing.

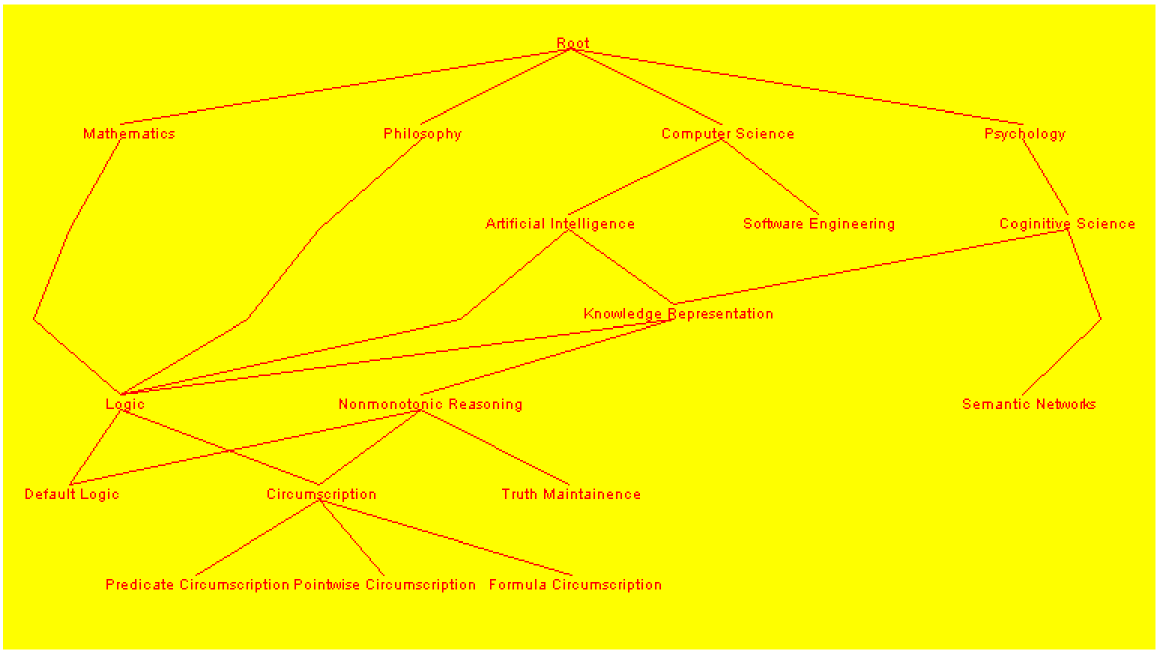

Figure 5.4: An output from a program that implements "permutation heuristic" with three sweeps, downward, upward, and then downward

This made it clear that one needs to give the layer-by-layer-sweep more time to spend traversing the graph. Two methods were considered for dealing with this. First, one can let the algorithm run for at least some given amount of time. This has the drawback, however, that it depends heavily on the size of the graph, and the traversal time differences can be very large. To illustrate: the average time for one traversal for the graph of Figure 5.1 is 30 milliseconds, while the average time to traverse the same graph with two more vertices added to the 4th layer is 10 seconds. Thus, if we specify that the program should run for at least five seconds, this turns out to be too long for the first graph and too short for the other one. Moreover, it would be difficult to choose an optimal time based on the size of the graph. For these reasons this author devised an alternative way to implement the algorithm.

The alternative allows the layer-by-layer-sweep to continue its work until the number of times each sweep does not produce any refinement reaches a certain number. This is herein called the "forgiveness number". A typical choice for this might be 20. The larger the number the more computation time is required.

Randomness is Needed

Another problem arose during a one-sided 2-level crossing reduction step. To illustrate, let us start at the "root" vertex and traverse downward. First we fix layer 1, in this case containing only the “root” vertex, and try to re-order layer 2 so that the number of crossings between layers 1 and 2 is minimum. This situation is shown in Figure 5.5.

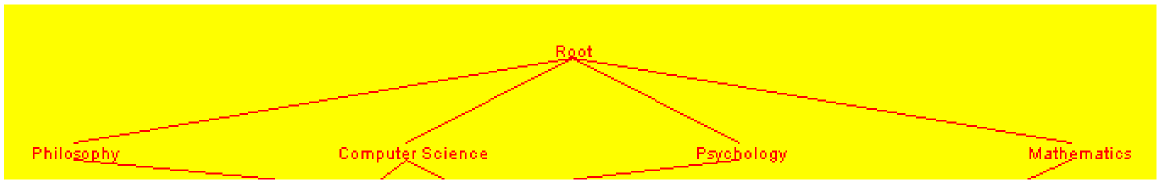

Figure 5.5: A display of layer 1 and layer 2 from the initial graph

In this topmost section of the graph there obviously cannot be any edge crossings regardless of the ordering of layer 2. Since this layer has four vertices, there can be 24 orderings (4-factorial) that can be the output ordering of this layer for this sweep. From the standpoint of the 2-level permutation heuristic, all these orderings are equivalent.

In practice, however, how one orders the vertices at higher layers can affect the ability to perform crossing reduction at lower layers. This is illustrated in Figures 5.6 and 5.7. The former shows the result of starting with the initial graph and performing one downward traversal. This yields a graph with three edge crossings. The latter shows the result of interchanging “Philosophy” and “Psychology” in the initial graph and again performing one downward sweep. This yields a graph with only one edge crossing.

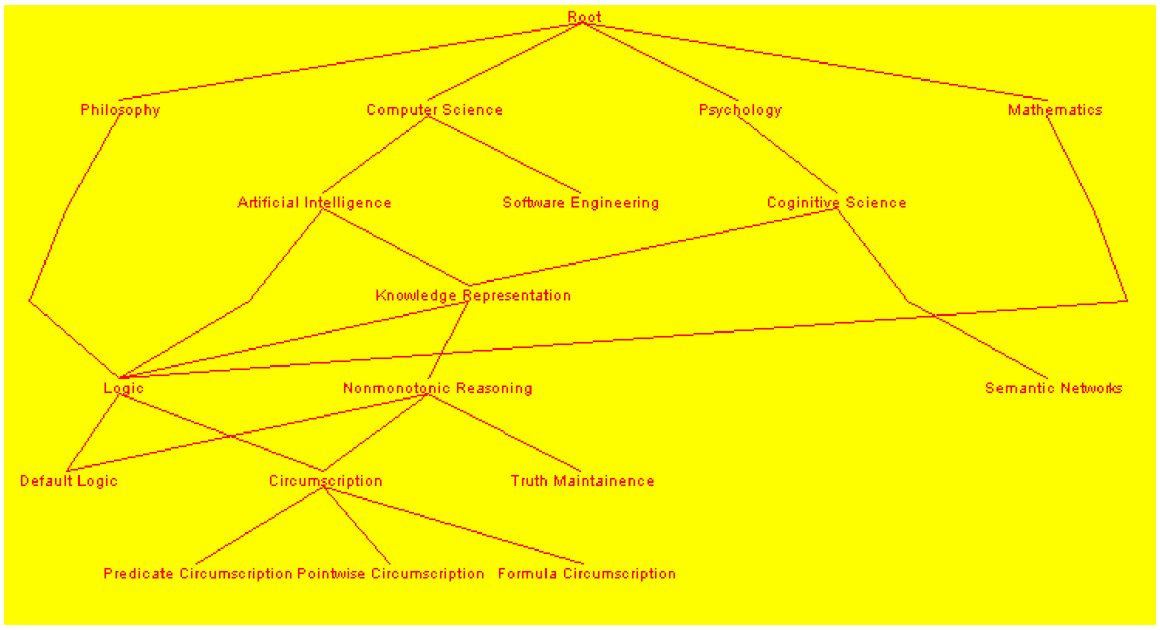

Figure 5.6: Output from one downward sweep applied to the initial graph

Nonetheless, as the sweep traverses back upward, the ordering of layer 2 will change, and another downward sweep may produce a better result. This was shown, in fact, in the sequence from Figure 5.2 to Figure 5.4.

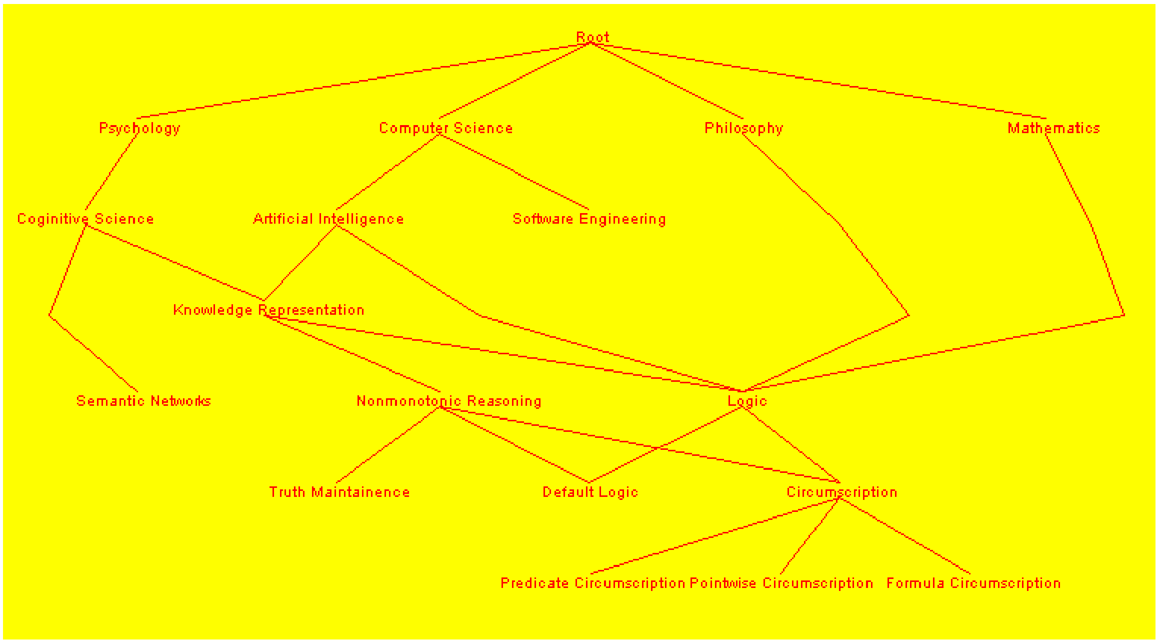

Figure 5.7: Output from one downward sweep applied to the initial graph modified by interchanging "Philosophy" and "Psychology"

However, after numerous further tests with different input graphs, it was found that this is not generally the case. It can in fact happen that the results of repeated sweeps will oscillate. For example, a downward sweep followed by an upward sweep can return the exact same graph with which it started.

This problem was fixed by an introduction of randomness. Let us start again from traversing downward by fixing layer 1 and try to re-order layer 2. Before applying the permutation heuristic to these two layers, we first randomly permute the vertices on layer 2. This has the effect of almost always returning a different graph than the one with which it started. This same technique is furthermore applied at all successive layers in the sweep. Moreover, this same technique is applied also in all upward sweeps.

This new implementation significantly improves the output of the program. The output graphs vary, however, inasmuch as there can be more than one drawing with minimal number of edge crossings.

The permutation heuristic has a significant drawback in that it is very slow. At each level, all possible permutations are explored. Thus if there are n vertices on a layer, n! permutations must be computed. The experiments showed that this already becomes impractical with n > 6. The results of various time comparisons are reported in Chapter 7.

Final Version of Permutation Heuristic Implementation

The final version of this implementation adds one more type of randomness. This is added to the permutation heuristic part, in contrast with the above, which added some randomness before applying the heuristic. The permutation heuristic is implemented as a recursive function that finds all permutations of a given layer. Let assume the layer has three vertices XYZ. The heuristic will find 3! = 6 orderings, namely, XYZ, XZY, YXZ, YZX, ZXY, and ZYX. For each permutation, the number of edge crossings for that particular pair of layers is computed. Assume that the ordering XYZ produces five crossings. This number will be assigned as the lowest_crossing_value, since it is the first ordering of the recursive function. The function will then find a second permutation, in this case XZY, and compute its number of edge crossings. Say this is 3. Then, since 3 < 5, this will be assigned as a new value for the lowest_crossing_value. On the other hand, if the XZY permutation has more than five edge crossings, this permutation is disregarded. This continues until all possible permutations have been compared.

The problem with this is that, when a there is more than one permutation with this same lowest_crossing_value number of edge crossings, only the first one is used. All subsequent ones are discarded. Since this may not necessarily lead to the best overall result, it was decided to introduce a random selection among these permutations in the following manner. If the number of crossing for a particular permutation is the same as the current value of the lowest_crossing_value, then this permutation will be used in place of the previous one with a 50% probability. With this additional modification, noticeably better results are achieved.

The summary of the implementation of the permutation heuristic is as follows:

1) The forgiveness number is initialized to 20, and a lowest_global_crossing variable is initialized to a very large number. The latter represents the total number of edge crossings in the entire graph.
2) A layer-by-layer sweep is applied, starting from the top-most layer by traversing downward. After the sweep reaches the bottom, it traverses back upward until it reaches the top.
3) For the downward traversal, the sweep starts by fixing layer 1 and using the "permutation function" (which implements the permutation heuristic described earlier) to re-order layer 2. Then it fixes layer 2, and applies the "permutation function" again to re-order layer 3. And so on. The downward traversal finishes when a bottom-most layer is re-ordered. For an upward traversal, the sweep starts by fixing layer k (bottom-most layer) and tries to re-order layer k-1. The sweep continues until layer 1 is re-ordered.
4) During a traversal, before the layer to be re-ordered is passed into the permutation function, the layer is randomly permuted.
5) In the permutation function, all possible orderings for the layer to be re-ordered are computed. For each permutation, the number n of edge crossings between this layer and the fixed layer is calculated. A variable lowest_crossing_value is initialized with the value n resulting from the first permutation. For each successive permutation, if n is less than the current lowest_crossing_value, the output ordering is set to that permutation and the lowest_crossing_value is set to n. If the number of edge crossings is equal to the lowest_crossing_value, there is a 50% chance that an output ordering will be set to that permutation.

6) During each traversal, the number of crossings for each particular two layers is added to a global_crossing variable. At the end of each traversal, this value is compared to a lowest_global_crossing in order to decide whether the sweep has produced any better result. If it has, the current ordering for the entire graph is copied to a data structure that will contain the final output, and the lowest_global_crossing is set to this value. If the traversal does not reduce the total number of crossings for the entire graph, the "forgiveness number" is reduced by one.
7) The traversal continues until the "forgiveness number" reaches zero. The final ordering for the entire graph is now in the data structure mentioned in step 6.

An Implementation of Barycenter Heuristic

General Idea

This algorithm uses the layer-by-layer-sweep with the barycenter heuristic in place of the permutation heuristic. The algorithm is the same as described above, except for steps 3, 4 and 5. Step 4 is removed, since randomness is applied in this implementation in a different way. In steps 3 and 5, we will use barycenter heuristic instead of permutation heuristic. As described in Chapter 4, this is a totally different idea from that employed by the permutation heuristic. The latter tries every possible permutation and compares their results. The barycenter heuristic only needs to calculate barycenter values for each vertex, and then sorts the vertices according to these values. Hence, no comparison between numbers of crossings is made.

Randomness is Also Needed

Assume that after we sort a layer we get the output shown in Table 5.1. There are two vertices that have a barycenter value of 3, and two vertices with a barycenter value of 5. If the randomness is not implemented, the output ordering of this layer will always be the one shown. For the same reasons as discussed with respect to the permutation heuristic, this is not always the best choice. Thus an element of randomness is added here into the comparison step of the sorting process. Then the output can be one of four possible orderings, since C and D can be switched, and so can G and H.

Table 5.1: Barycenter value and name of vertex

Barycenter value	1	2.5	3	3	3.5	4.5	5	5
Name of vertex	A	B	C	D	E	F	G	H

Experiments with this idea on the aforementioned initial graph showed that the barycenter heuristic with randomness gives a best result of one crossing, while the one without randomness has three crossings. Similar results were obtained with other graphs.

Final Version of Barycenter Heuristic Implementation

An implementation of this heuristic is described below.

1) The forgiveness number is initialized to 20, and a lowest_global_crossing variable is initialized to a very large number. The latter represents the total number of edge crossings in the entire graph.
2) A layer-by-layer sweep is applied, starting from the top-most layer by traversing downward. After the sweep reaches the bottom, it traverses back upward until it reaches the top.
3) For the downward traversal, the sweep starts by fixing layer 1 and using the "barycenter function" (which implements the barycenter heuristic described earlier) to re-order layer 2. Then it fixes layer 2, and applies the "barycenter function" again to re-order layer 3. And so on. The downward traversal finishes when a bottom-most layer is re-ordered. For an upward traversal, the sweep starts by fixing layer k (bottom-most layer) and tries to re-order layer k-1. The sweep continues until layer 1 is re-ordered.
4) During the re-ordering process, the "barycenter function" calculates the barycenter value for each vertex. All the vertices are then sorted by barycenter values. Vertices with lower barycenter values will be at the beginning, while vertices with higher values will be at the end. During the sorting process, if two barycenter values are the same, there is a 50% chance that the two vertices will be switched.
5) During each traversal, the number of crossings for each particular two layers is added to a global_crossing variable. At the end of each traversal, this value is compared to a lowest_global_crossing in order to decide whether the sweep has produced any better result. If it has, the current ordering for the entire graph is copied to a data structure that will contain the final output, and the lowest_global_crossing is set to this value. If the traversal does not reduce the total number of crossings for the entire graph, the "forgiveness number" is reduced by one.
6) The traversal continues until the "forgiveness number" reaches zero. The final ordering for the entire graph is now in the data structure mentioned in step 5.

CHAPTER 6

A HYBRID METHOD FOR K-LEVEL CROSSING REDUCTION

A Hybrid Between Permutation Heuristic and Barycenter Heuristic

As described in an earlier chapter, the barycenter heuristic is the most popular heuristic for crossing reduction. However, the number of crossings produced by the barycenter heuristic can be as bad as $\Omega(\sqrt{n})$ times the optimum. From the fact that a permutation heuristic is only good for graphs where each layer has a small number of vertices, a combination of these two heuristics was explored. The new implementation combines these two heuristics together based on the layer-by-layer-sweep algorithm described in Chapter 4. When the sweep algorithm tries to re-order each layer, it first finds the number of vertices of that layer. If this number is small, the layer is re-ordered using the permutation heuristic. Otherwise, the barycenter heuristic is used.

From an experiment with the permutation heuristic, discussed in the following chapter, we find that there is a large difference in the time used for a graph with a width of 6 vertices and that for a graph with a width of 7 vertices. Hence, the new heuristic uses the permutation heuristic to re-order layers with 6 vertices or fewer and uses the barycenter heuristic otherwise. The "barycenter number" is the number that determines which method to be used, and it is set to 6.

Algorithm for the New Method

This new method can be described as follows.

1) The forgiveness number is initialized to 20, and a lowest_global_crossing variable is initialized to a very large number. The latter represents the total number of edge crossings in the entire graph.
2) A layer-by-layer sweep is applied, starting from the top-most layer by traversing downward. After the sweep reaches the bottom, it traverses back upward until it reaches the top.
3) For the downward traversal, the sweep starts by fixing layer 1 and re-order layer 2. Then it fixes layer 2, and re-order layer 3. And so on. The downward traversal finishes when a bottom-most layer is re-ordered. For an upward traversal, the sweep starts by fixing layer k (bottom-most layer) and tries to re-order layer k-1. The sweep continues until layer 1 is re-ordered.
4) For each layer to be re-ordered, the sweep decides whether it should use the barycenter heuristic or the permutation heuristic. If the layer contains 6 vertices or fewer, the permutation heuristic is used; otherwise, the barycenter heuristic is used.
5) If the permutation heuristic is used for a layer, this layer is first randomly permuted before being passed into the permutation function. This is the same as

for steps 4 and 5 in an implementation of permutation heuristic. If the barycenter heuristic is used, the layer is simply passed to a “barycenter function”.

6) During each traversal, the number of crossings for each particular two layers is added to a global_crossing variable. At the end of each traversal, this value is compared to a lowest_global_crossing in order to decide whether the sweep has produced any better result. If it has, the current ordering for the entire graph is copied to a data structure that will contain the final output, and the lowest_global_crossing is set to this value. If the traversal does not reduce the total number of crossings for the entire graph, the “forgiveness number” is reduced by one.
7) The traversal continues until the “forgiveness number” reaches zero. The final ordering for the entire graph is now in the data structure mentioned in step 6.

CHAPTER 7

COMPUTATIONAL RESULTS

Implementations Used in this Test

This thesis compares results from the five crossing reduction implementations discussed in the foregoing:

- Permutation without randomness
- Permutation with randomness
- Barycenter without randomness
- Barycenter with randomness
- Hybrid, permutation plus barycenter

Results for a Graph with Layers Containing up to 5 Vertices

Let us start with the initial graph shown in Figure 7.1 (the same as Figure 5.1).

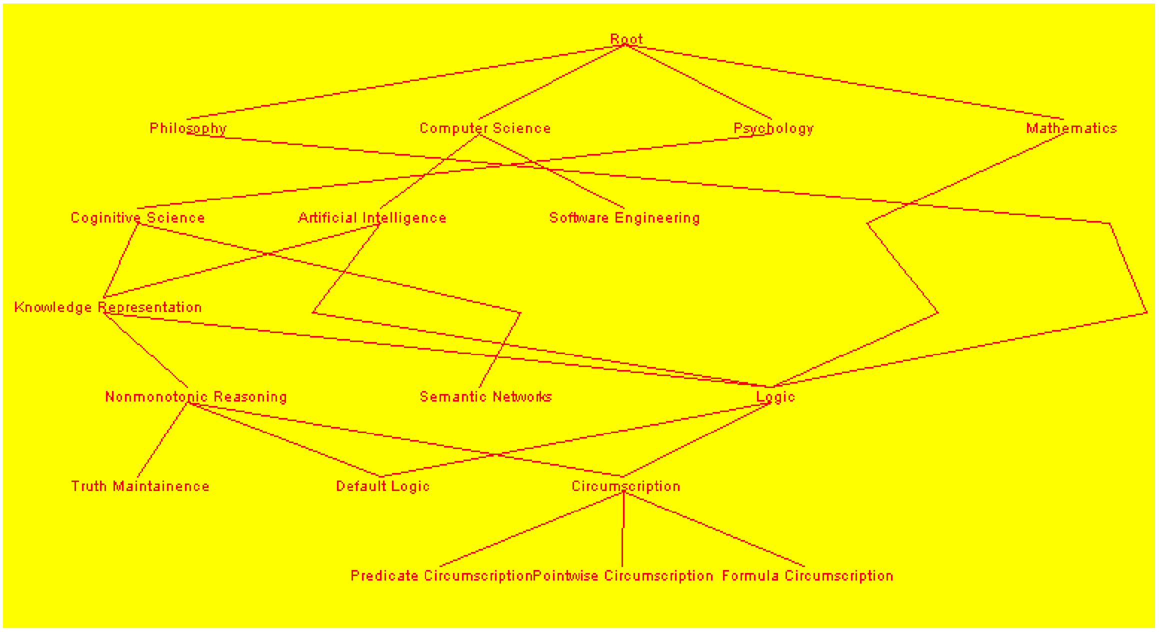

Figure 7.1: The initial graph

The “barycenter without randomness” is the only approach that does not produce an optimal result. Result from the “permutation without randomness” approach is shown in Figure 7.2. Results from the other three implementations vary since randomness is involved. One of an output from the “barycenter with randomness” approach is shown in Figure 7.3. This graph can also be a result from the other two randomized methods.

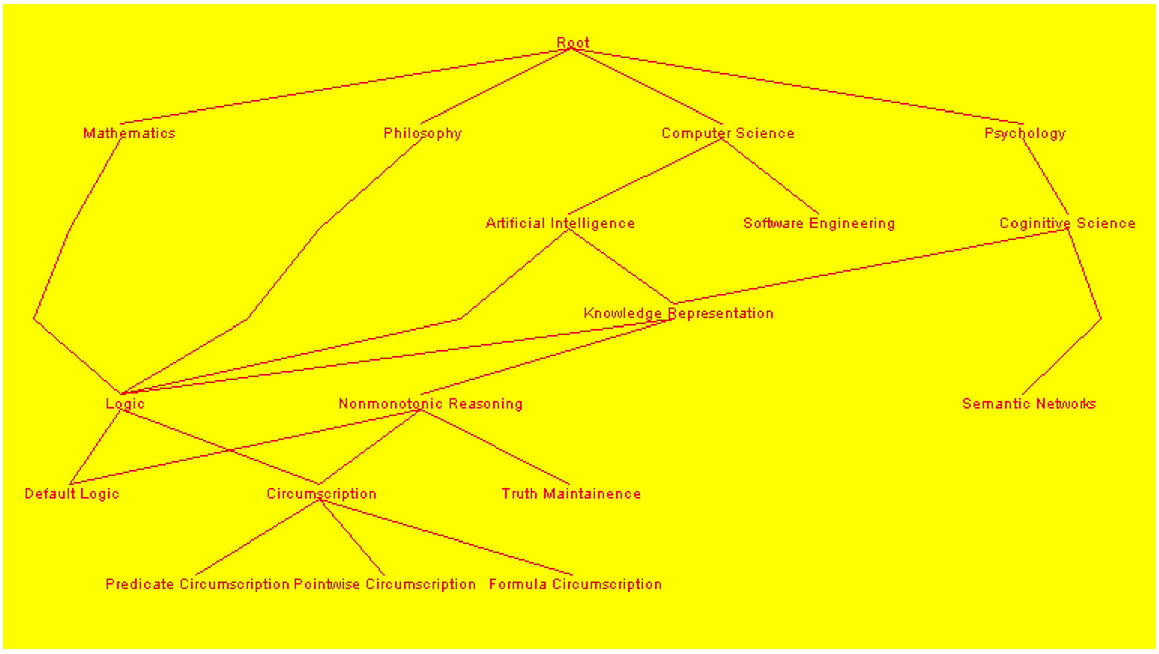

Figure 7.2: Result from the permutation without randomness approach

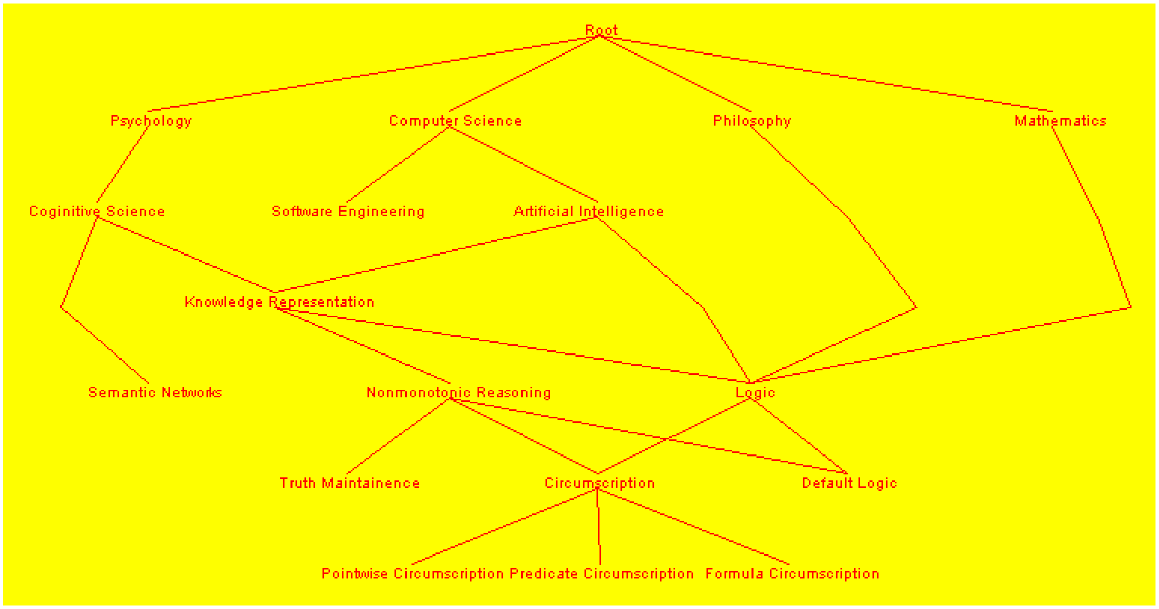

Figure 7.3: A result from the barycenter with randomness approach

The result for the “hybrid” approach for this graph is more or less the same as the result for the “permutation with randomness” approach, since there is no layer with seven or more vertices. An example in the next section adds a few more vertices to the graph.

The average times used for this input are shown in Table 7.1. An “R” at the end of an implementation name in the table represents “randomness”.

Table 7.1: Time comparison in millisecond for an input graph with a width of 5 vertices

Implementation	Permute	Permute-R	Bary	Bary-R	Hybrid
Time Used	710	725	50	50	725

Results for a Graph with Layers Containing up to 8 Vertices

For purposes of displaying larger graphs on the screen, in this section and the next we use numbers as labels for vertices. A new input graph for experimentation is shown in Figure 7.4. This differs from the initial graph in Figure 7.1 only by adding three more vertices (19, 20, and 21) to the fourth layer and connecting them to vertex 5, which in Figure 7.1 is “Mathematics”. This makes layers 3 and 4 both have 8 vertices. The average times of the five implementations when applied to this graph are shown in Table 7.2

Table 7.2: Time comparison in millisecond for an input graph with a width of 8 vertices

Implementation	Permute	Permute-R	Bary	Bary-R	Hybrid
Time Used	6671000	6934000	60	60	130

This shows, in particular, that the permutation heuristic, whether with or without randomness, is already impractical for a layer with only eight vertices---both versions took almost two hours to compute. If the input width is reduced by one, to a width of seven vertices, these two algorithms take approximately four minutes. For a width of six vertices, they take approximately 10 seconds. Hence, the “barycenter number” in the “hybrid” implementation should be set to 6.

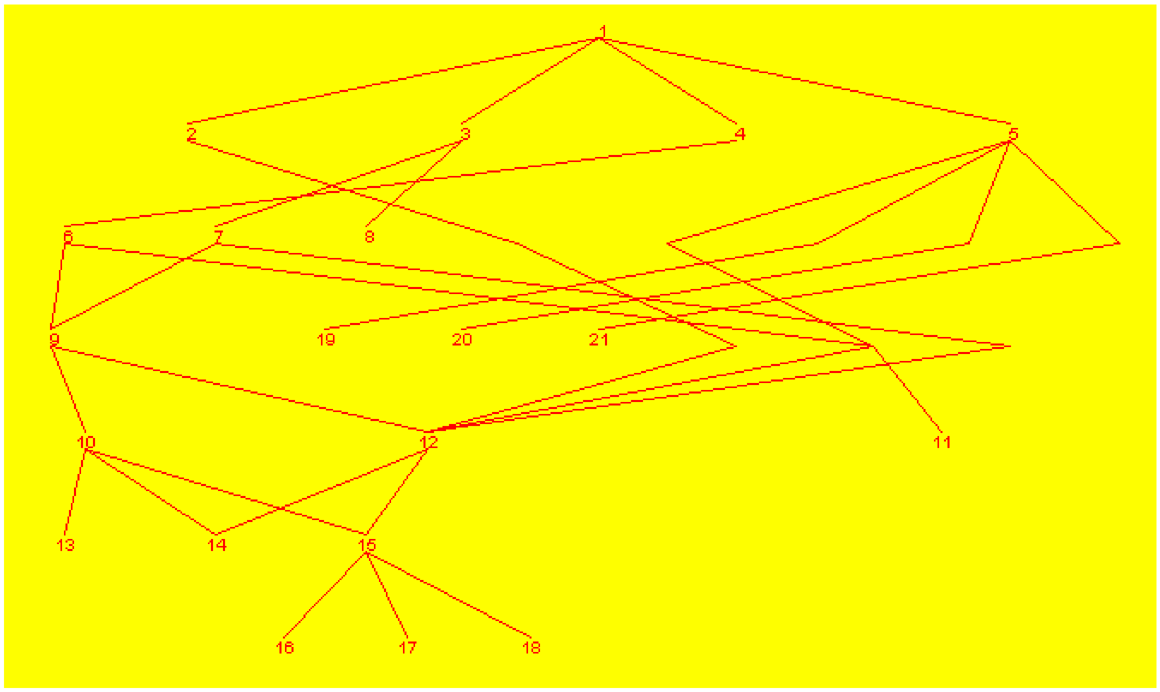

Figure 7.4: The new input graph with a width of 8 vertices

Some results of applying the five algorithms to the graph in Figure 7.4 are shown in Figures 7.5 through 7.9. Note that for the three algorithms that involve randomness, different runs will tend to produce different output graphs. This time, the "permutation without randomness" and the "barycenter without randomness" algorithms did not produce an optimal result, while the other three did.

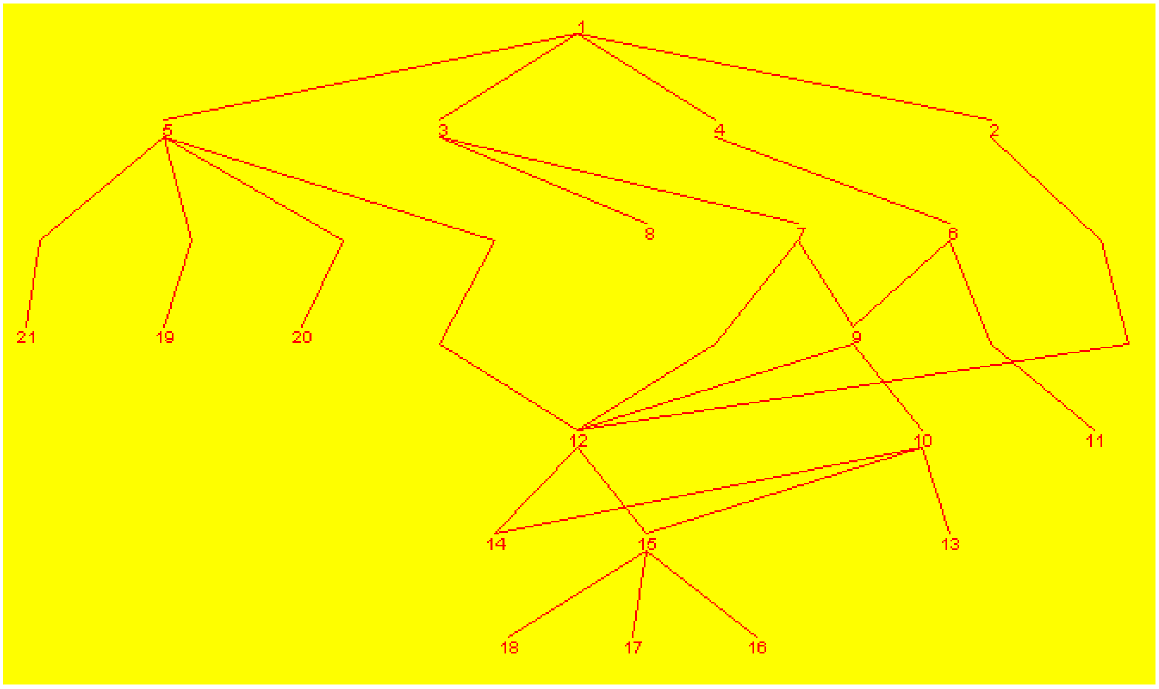

Figure 7.5: Result from the permutation without randomness approach

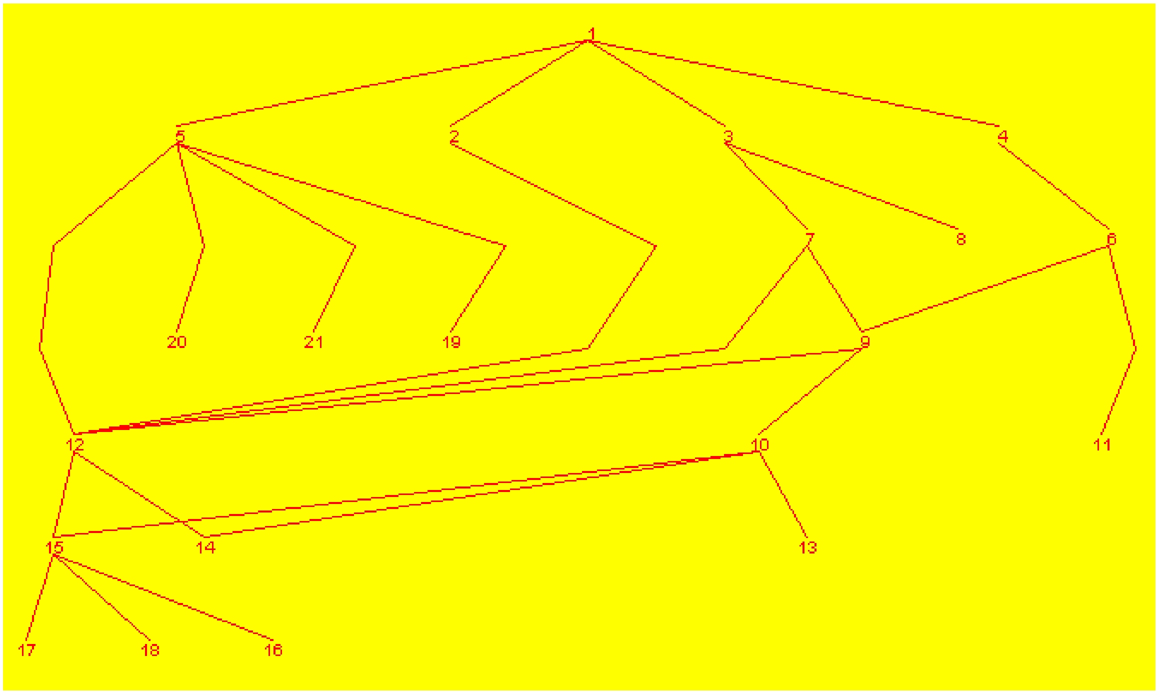

Figure 7.6: A result from the permutation with randomness approach

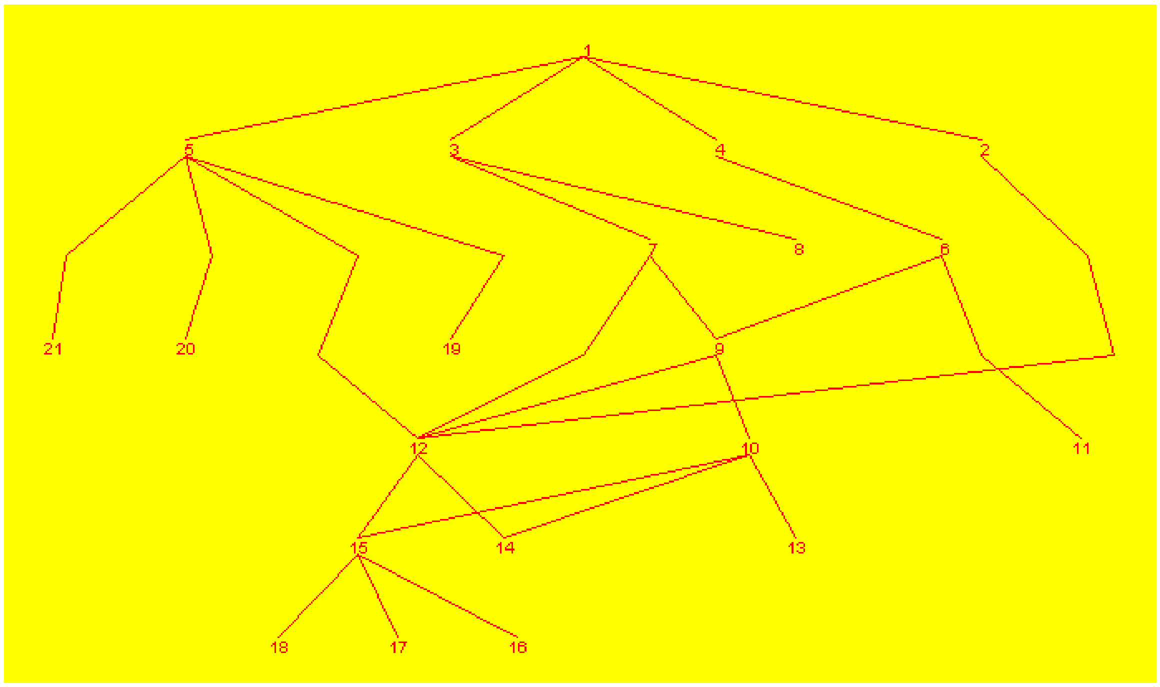

Figure 7.7: Result from the barycenter without randomness approach

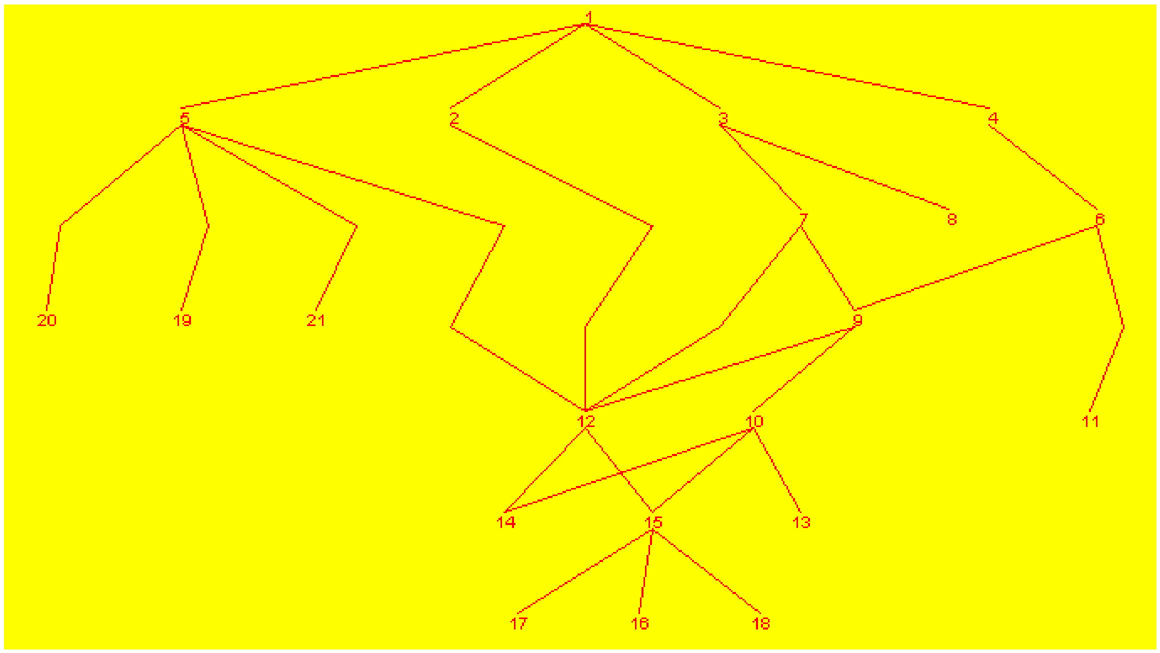

Figure 7.8: A result from the barycenter with randomness approach

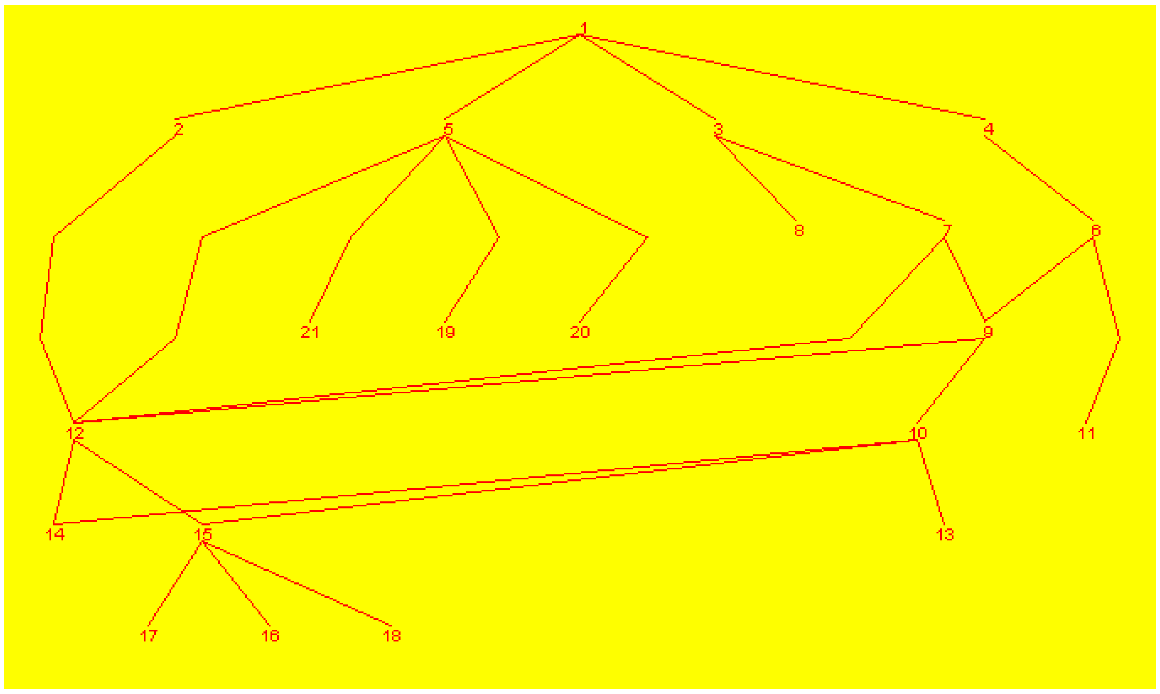

Figure 7.9: A result from the hybrid approach

Results for a Graph with Layers Containing up to 35 Vertices

The results reported in this section use the new input graph shown in Figure 7.10. This graph has a width of 35 vertices.

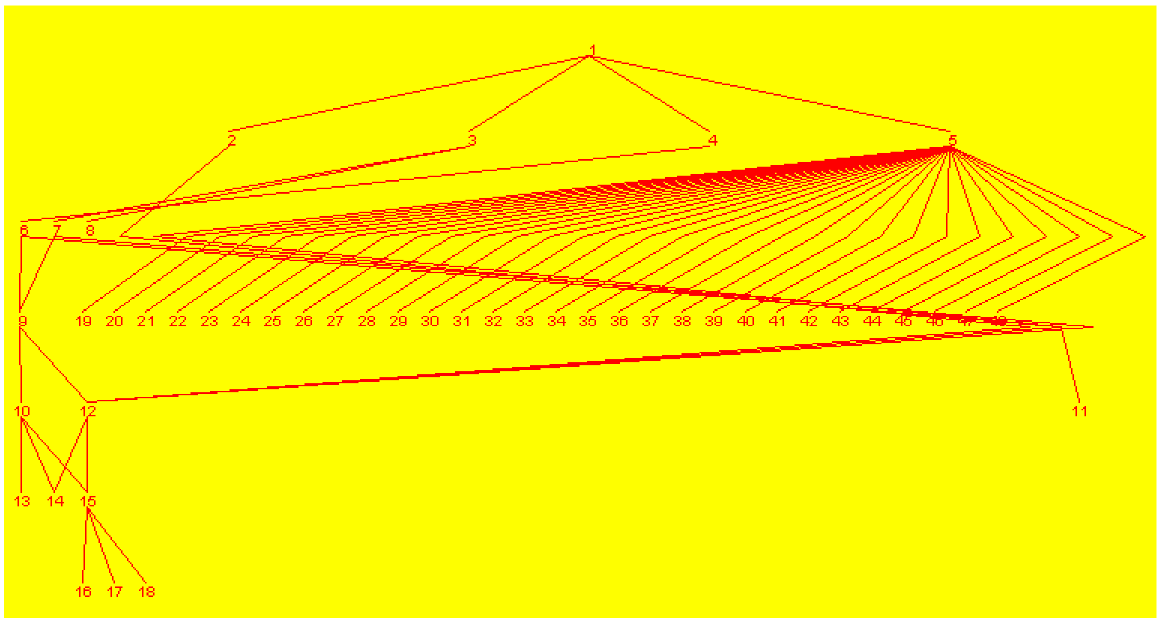

Figure 7.10: The new input graph with a width of 35 vertices

Because of its width, neither permutation heuristic can be used. Results from one run of each of the "barycenter with randomness" heuristic and the "hybrid" approach are shown in Figures 7.11 and 7.12.

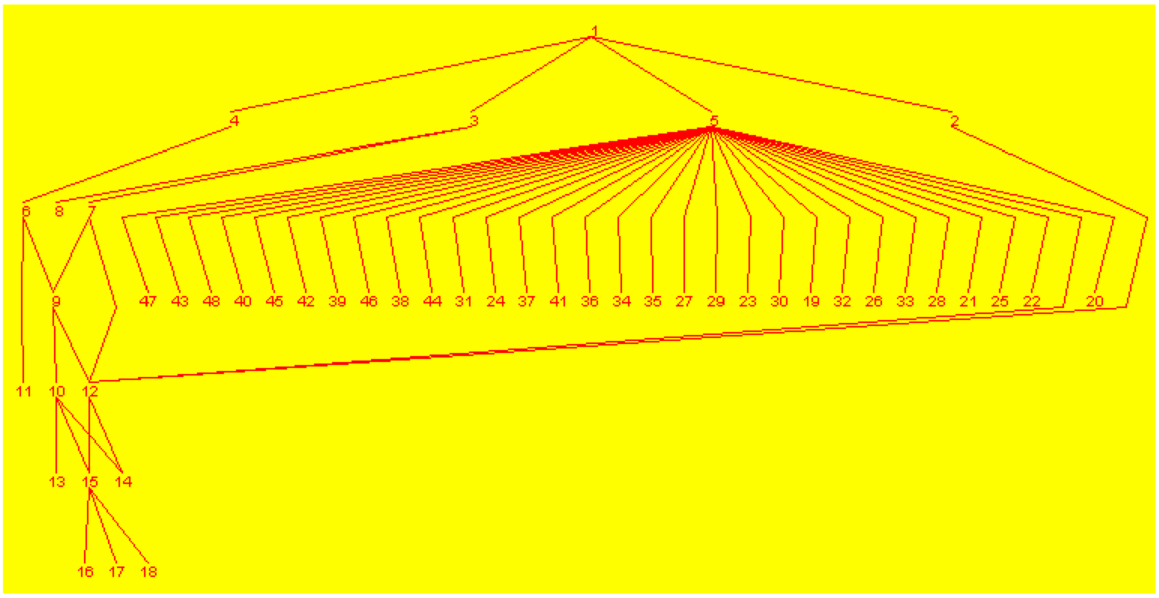

Figure 7.11: A result from the barycenter with randomness approach

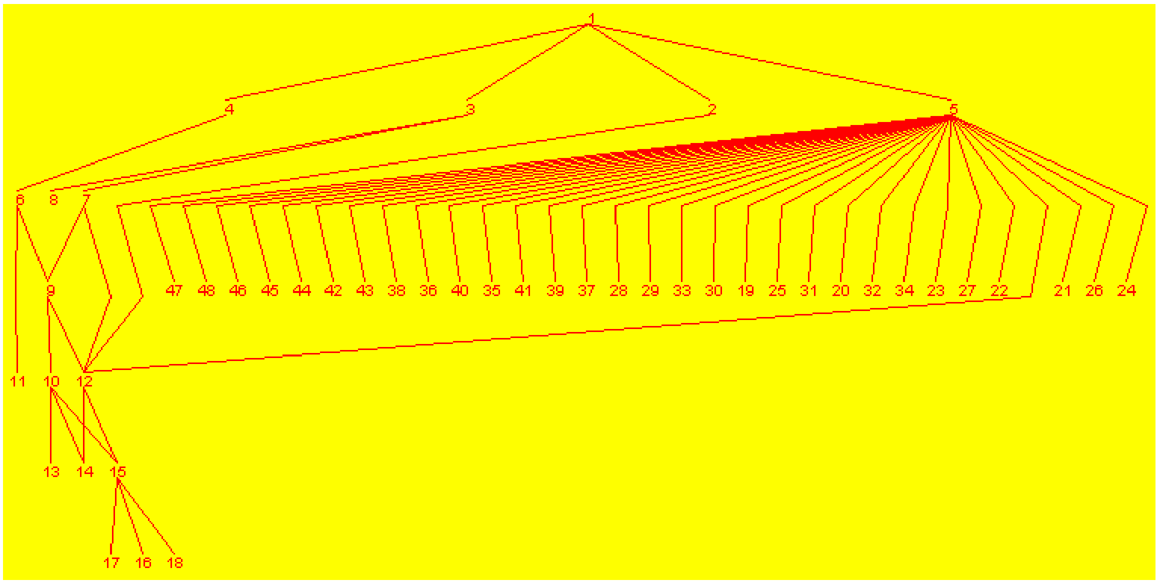

Figure 7.12: A result from the hybrid approach

The following two tables show the numbers of edge crossings for each traversal for ten runs of each algorithm (barycenter with randomness and hybrid). Consider the first row, labeled "Run 1" in Table 7.3. The number 4 in the second column is the total number of crossings after the first traversal of the layer-by-layer-sweep. In this case, it is the end of the first downward traversal. Since this is a first traversal, the lowest_global_crossing is initialized to 4 and the current configuration of the entire graph is saved. After this downward traversal, the sweep continues traversing upward until it reaches the top layer.

The total number of edge crossings at this point is 40. This might seem to be very large compared to the four crossings achieved previously, but this is a nature of the "layer-by-layer-sweep". Since this considers only two layers at a time, it cannot anticipate all the crossings for the entire graph. If one lets the sweep process to continue repeating long enough, one may expect this to eventually achieve the best result that the given algorithm can attain. In practice, however, it is necessary to set a limit of how long the sweep process may continue running.

It was for this reason that the notion of "forgiveness number", discussed in Chapter 5, was introduced. One starts each run with this number set to some value (in these cases this value is 20) and then this number is reduced by 1 each time a traversal (either downward or upward) fails to produce a smaller number of crossings than the best result attained so far (which is stored in lowest_global_crossing). Thus, for example, after the second traversal in Run 1, since 40 are greater than 4, the "forgiveness number" is reduced from 20 to 19.

Table 7.3: Results from each traversal from the barycenter with randomness algorithm

Traversal	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**
Run 1	4	40	33	35	**1**	36	34	36	3	35	6	4	**1**	2	**1**	4	34	5	32	38	6	37	
Run 2	33	37	4	42	34	35	4	38	**1**	2	**1**	3	3	38	32	36	4	38	6	39	2	36	6
Run 3	6	38	32	35	4	4	7	42	**1**	2	4	36	4	37	**1**	42	33	36	3	38	2	39	33
Run 4	4	41	3	38	**1**	**1**	4	41	4	37	**1**	39	2	38	**1**	35	32	38	6	43	4	38	4
Run 5	**1**	2	32	37	4	42	7	43	4	38	3	36	4	6	34	33	6	38	4	39	33		
Run 6	**1**	**1**	4	41	**1**	36	2	37	3	37	32	37	32	41	3	3	33	36	4	41	4		
Run 7	2	36	4	44	5	39	4	39	**1**	39	2	42	4	45	4	38	3	39	4	40	3	2	
Run 8	4	38	6	43	3	3	4	43	4	41	**1**	41	34	36	3	40	3	36	3	37	4	39	4
Run 9	**1**	38	**1**	39	**1**	40	4	38	3	42	33	36	4	38	33	37	**1**	38	**1**	42	6		
Run 10	3	2	34	13	33	35	32	34	32	38	3	40	4	37	4	40	34	11	2	41	6	4	

Table 7.4: Results from each traversal from the hybrid algorithm

Traversal	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**	**21**	**22**	**23**	**24**
Run 1	34	5	**1**	38	3	6	3	43	**1**	7	4	40	33	38	3	4	34	35	**1**	40	34	33	3	
Run 2	32	36	32	38	**1**	43	2	6	34	37	**1**	38	**1**	38	33	4	34	8	3	40	34	36		
Run 3	34	10	3	6	34	6	3	38	**1**	**1**	34	6	34	10	4	37	3	9	34	35	4	9	33	7
Run 4	4	41	4	2	3	45	3	41	34	34	34	11	34	9	33	37	**1**	5	34	39	34	40	**1**	
Run 5	4	5	4	5	34	34	4	40	3	38	34	10	34	8	3	43	33	4	**1**	7	34	38	3	
Run 6	4	4	4	8	33	8	4	40	33	39	4	38	4	7	3	39	**1**	4	**1**	6	**1**	39	**1**	
Run 7	4	42	4	38	33	35	32	46	**1**	8	32	38	3	6	**1**	4	3	42	34	10	4	42		
Run 8	33	8	**1**	41	4	42	4	38	4	3	34	33	3	37	34	13	3	41	3	42	**1**	10	4	
Run 9	34	11	4	37	**1**	40	34	6	34	8	3	9	3	5	3	40	34	11	4	**1**	4	3	34	36
Run 10	3	44	**1**	3	34	10	**1**	4	4	7	4	38	3	7	34	11	3	40	4	5	3	44		

The third and fourth traversals of Run 1 in Table 7.3 respectively produce 33 and 35 crossings, both of which are also more than 4, so that at this point the forgiveness number is 17. The fifth traversal gives 1 crossing, however, which is better than 4. Hence, now the lowest_global_crossing is set to 1 and the current configuration of the graph is saved.

Here an issue arises regarding whether we should at this point set the "forgiveness number" back to 20. It does make sense to let the process run for 20 more traversals after each time it produces a better result. This would be a reasonable strategy, and it is very easy to change in the code to accommodate this. However, it was decided to let the number stay as before (e.g., currently the value is 17). The main reasons for this were that (i) this makes the different runs all take about the same time, and (ii) experiments showed that the results of the two strategies are not very much different.

In Run 1 of Figure 7.3, after the 22nd traversal the "forgiveness number" is reduced to zero, and the procedure stops. So the final output ordering in this run is the output of the fifth traversal, which is a third downward traversal.

The traversals that produce the best results are highlighted in the table. For the given input graph, the best result is a graph with one crossing.

From the two tables we see that the barycenter with randomness approach failed to produce the best possible result for one of the ten runs (Run 10), while the "hybrid" approach produced the best possible result for all runs. Further experimentation showed, however, that for 20 runs both implementations failed to produce best result for one run. This means that, in general, the two algorithms may be equally effective.

For the results reported in Tables 7.3 and 7.4, both algorithms used less than 1 second to compute each run, but the hybrid algorithm did use approximately 45% more time than the barycenter with randomness algorithm.

Several further experiments were tried with these same two algorithms. The hybrid produces slightly better results overall than the barycenter with randomness method. To illustrate, consider the input graph shown in Figure 7.13. For this, the "barycenter with randomness" produced a best result for 12 out of 20 runs, while the "hybrid" produced a best result for 19 out of 20 runs. The output of one such "hybrid" run with best result (3 crossings) is shown in Figure 7.14.

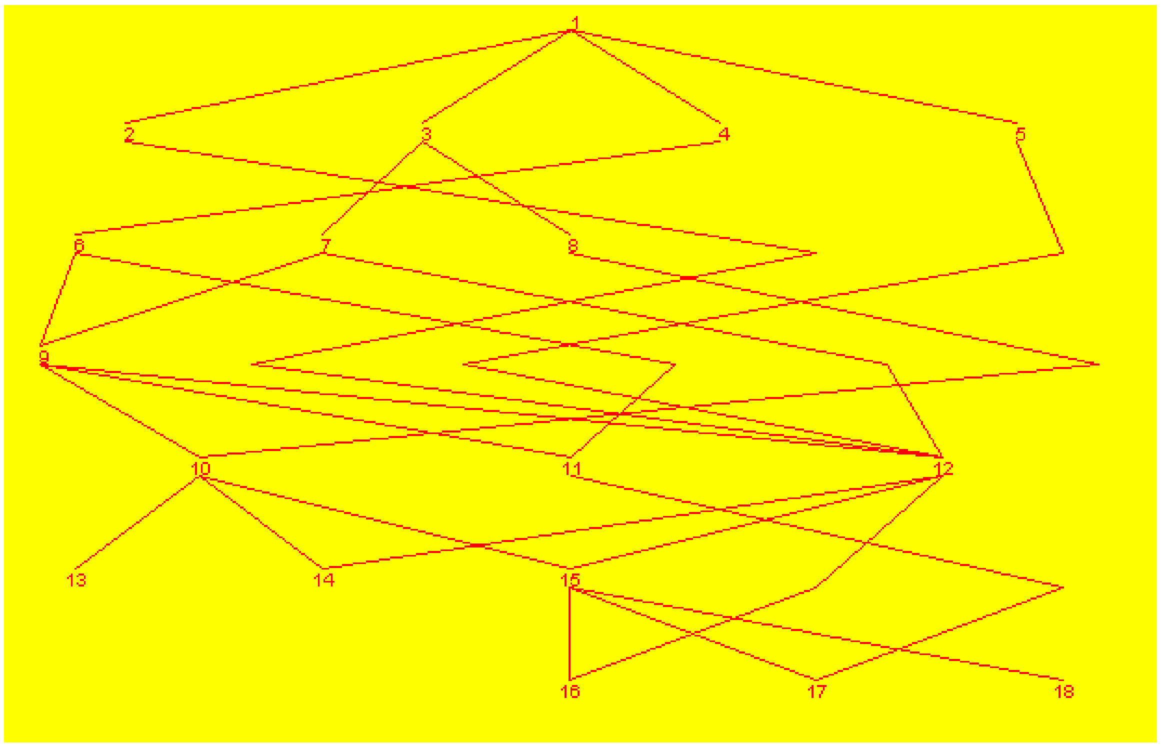

Figure 7.13: The new input graph

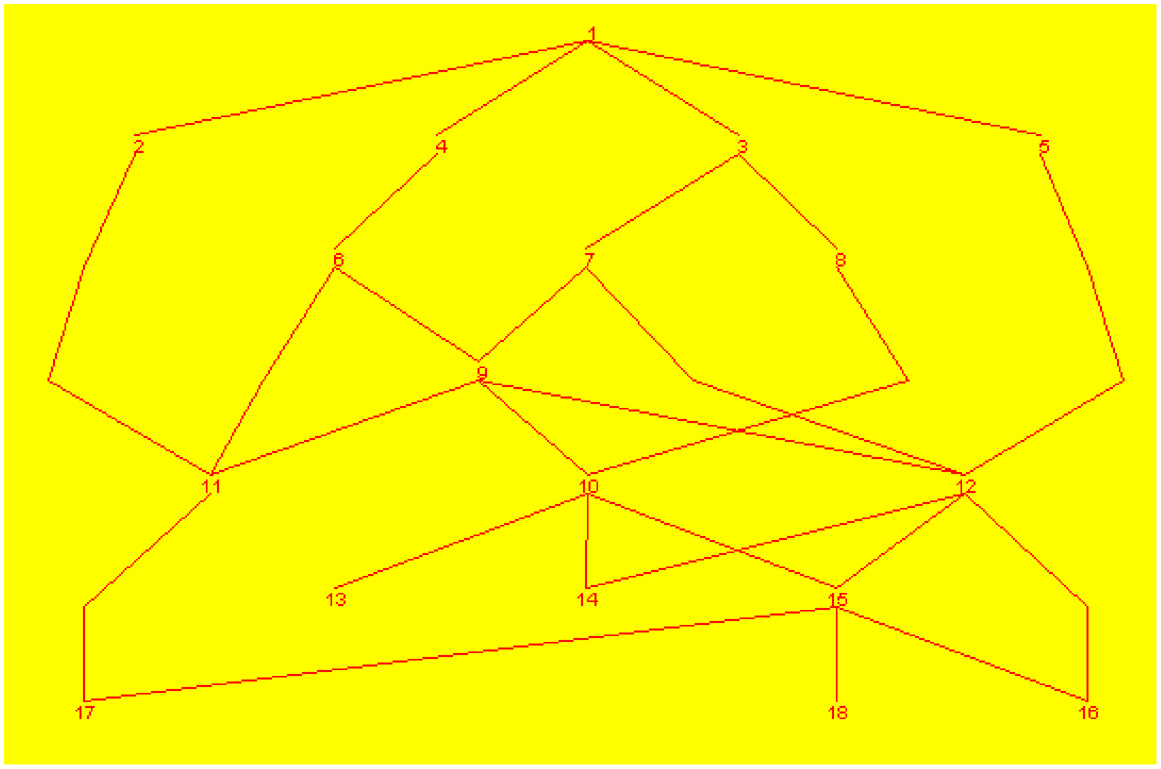

Figure 7.14: A result from the hybrid approach

If we increase the forgiveness number of the randomized barycenter implementation from 20 to 200, however, then this algorithm produced a best result for all of the first 20 runs.

If we similarly change the forgiveness number of the hybrid implementation from 20 to 40, then again we obtain a best result in all of the first 20 runs. This suggests that, for relatively uncomplicated graphs, if we let both algorithms run long enough, they will attain the best possible result.

For a very large complicated graph, however, it may be a case that neither approach will give an optimal result no matter how long they run. These kinds of graphs are beyond the scope of this thesis. Since the "barycenter number" (maximum number of vertices for applying the permutation heuristic in the hybrid approach) in this research is small, one may expect the results of the two approaches to be about the same, or possibly that the hybrid method might be slightly better. A further suggested method will be mentioned in Chapter 8. It is conjectured that this method will be able to perform better than any of those considered here.

Time Comparison

Table 7.5 shows a time comparison between the various implementations for graphs of various widths. These graphs all had 7 layers, and were essentially identical except for layers 3 and 4. In particular, the graphs of Figures 7.1, 7.4, and 7.10 were used respectively for the lines with "maximum layer width" of 5, 8, and 35. All other graphs varied only by having the indicated number of vertices placed into level 4, with dummy vertices as needed placed into layer 3.

In all cases, the algorithms were run with a "forgiveness number" of 20. Most results use averages from 20 runs. Exceptions were those that required more than one minute to calculate. Specifically, for the two permutation algorithms, we used four runs for the graph with seven vertices and two runs for the graph with eight vertices. These were all done on a machine with an AMD Athlon 1600+ CPU.

Table 7.5: Time comparison between various implementations in millisecond

Maximum Layer Width	Permute	Permute-R	Bary	Bary-R	Hybrid
5	710	725	50	50	725
6	9850	10350	55	55	10400
7	204500	212500	55	60	130
8	6671000	6934000	60	60	130
15			100	110	170
35			225	245	355

Note that, when a graph has a width of 6 vertices, the hybrid algorithm is slow (11 seconds). This is because it uses the permutation heuristic for such layers. Thus, if a graph has many layers of this width, this approach may be impractical.

In contrast, however, if the graph does not have any layers of width 5 or 6, but does have layers containing up to 35 vertices, then, even though the hybrid approach is somewhat slower than the randomized barycenter approach, its overall performance may be somewhat better. This is because, as noted in the previous section, the latter needs a larger forgiveness number to achieve the same results as the former. Thus, overall, the two approaches seem to be equally practical.

CHAPTER 8

FURTHER WORK

Combination of Branch-and-Cut Heuristic and Barycenter Heuristic

Junger and Mutzel [5] have implemented a "branch-and-cut" heuristic for one-sided 2-level crossing reduction and have shown this is practical to use for layers having up to 60 vertices. As with the permutation heuristic, it produces an optimal result for one-sided 2-level crossing reduction. It is much faster, however, since the permutation heuristic is bounded by n-factorial, while the branch-and-cut heuristic has a polynomial bound.

Thus a considerably better "hybrid" than that discussed in this thesis might be one that uses this heuristic in place of the permutation heuristic and sets the "barycenter number" to a higher value.

This hybrid approach was not explored in this thesis because the branch-and-cut algorithm is very complex and would take a long time to implement. The author believes that a combination of branch-and-cut heuristic and randomized barycenter heuristic promises very good result. This should be better than the barycenter method standing alone and still be fast enough for practical use.

More Samples Should be Tested

All samples used in this thesis are modified from a sample graph used by Reddy [9]. Several more input graphs should be explored. Furthermore, since the barycenter heuristic does not perform well for dense graphs, more samples of this kind of graph should also be tested.

CONCLUSION

There have been several studies that compare results and times for 2-level crossing reduction implementations. The studies for k-level crossing reduction implementation, however, are much fewer.

This thesis studied and implemented k-level crossing reduction for layered hierarchical graph drawing. It uses a general “layer-by-layer-sweep”, together with two heuristics: permutation and barycenter. There were several problems that had to be overcome, stemming from the fact that there were no available implementations and the literature describing the algorithms left out many details. These included how to decide when to stop the “layer-by-layer-sweep” procedure and the need to introduce randomness into both heuristics. Accordingly, the explanation in this thesis of how to implement these heuristics should be useful for others.

It was found that the permutation heuristic is impractical for moderate and large graphs, and the barycenter heuristic does not always produce good result. Accordingly, this thesis combined the permutation and the barycenter heuristics in order to produce a new implementation that may perform better than the barycenter implementation for large graphs. For layers with small width, the permutation heuristic is used; otherwise, the barycenter heuristic is used. The notion of “barycenter number” was introduced for this purpose. This is the number that decides whether a re-ordering layer uses the barycenter or permutation heuristic. In these experiments, this number was set to 6.

The tests showed that the hybrid implementation is fast enough to replace the randomized barycenter implementation, but it does not seem to provide any significant improvement over it. However, more input graphs should be tested. The barycenter heuristic tends to perform worse when an input graph is dense. Large dense graphs may be able to show significant improvement of the suggested method over the barycenter method.

This thesis showed that a hybrid method was worth considering, especially if one could find an effective hybrid method. At the end, a new kind of hybrid approach was suggested for future work. This would use a branch-and-cut heuristic in place of the permutation heuristic. It is conjectured that this will provide better results overall than any of the approaches considered here.

REFERENCES

[1] G. Di Battista, *Graph drawing: algorithms for the visualization of graphs*, Prentice Hall, 1999.

[2] M. Junger, P. Mutzel, 2-layer straightline crossing minimization: performance of exact and heuristic algorithms, *Journal of Graph Algorithms and Applications*, Vol. 1, 1997, pp. 33-59.

[3] G. Di Battista, P. Eades, R. Tamassia, I.G. Tollis, *Algorithms for drawing graphs: an annotated bibliography*, Report, Brown University, June 1994.

[4] M. Petra, W. Rene, Two layer planarization in graph drawing, K.Y. Chwa and O.H. Ibarra (eds.), *Proc. ISAAC '98, LNCS 1533*, Springer Verlag, 1998, pp. 69-78.

[5] M. Junger, P. Mutzel, Exact and heuristic algorithm for 2-layer straightline crossing number, *Proc. Graph Drawing '95, Lecture Notes in Computer Science 1027*, Springer Verlag, 1996, pp. 337-348.

[6] C. Matuszewski, R. Schoenfeld, P. Molitor, Using sifting for k-layer crossing minimization, J. Kratochvil (ed.), *Proc. Graph Drawing (GD '99), Volume 1731 of LNCS*, Springer-Verlag, 1999, pp. 217-224.

[7] P. Mutzel, An alternative method to crossing minimization on hierarchical graphs, *Proc. Graph Drawing '96, LNCS*, Springer Verlag, 1997, pp. 318-333.

[8] X.Y. Li, M.F. Stallmann. New bound on the barycenter heuristic for bipartite graph drawing, *Information Processing Letters*, 82 (2002) 293-298.

[9] N. Reddy, *An algorithm for knowledge representation using layered directed acyclic graphs*, Masters Project Report, Florida State University, Fall 2003.

BIOGRAPHICAL SKETCH

Pitch Patarasuk was born in Bangkok, Thailand, in 1977. He earned his high school degree from Triam Udom Suksa High School, which is generally considered the best high school in Thailand. In March 1999, he received a Bachelor of Engineering from Chulalongkorn University, Thailand, which is the first and the most excellent university in the country. He continued his master study at Florida State University in August 2000. During his master study he was selected as a member of Upsilon Pi Epsilon, the honor society in the computing and information disciplines.