A Hierarchical Layout Algorithm for Drawing Directed Graphs

by

Jason Reynolds

A thesis submitted to the
Department of Computer and Information Science
in conformity with the requirements for
the degree of Master of Science

Queen's University
Kingston, Ontario, Canada
April, 1997

Abstract

A significant problem with hierarchical layout drawing algorithms for directed graphs is the inability to draw large graphs well, in a reasonable amount of time. This paper describes an extension to the algorithm presented in [GKNV93] that improves efficiency dramatically. By adding a pre-processing step to the algorithm, a heuristic can be used to modify the structure of the initial input graph. These modifications automatically improve the efficiency of the algorithm in [GKNV93] while still producing hierarchical layouts of equal or better quality.

Thesis Supervisor: Dr. D. Rappaport
Title: Associate Professor

Table of Contents

List of Tables

List of Graphs

List of Formulas

List of Integer Programs

List of Figures

Chapter 1 - Introduction

Currently there are many applications that use directed graphs to represent information. ER diagrams show the relations in a database. PERT diagrams reveal dependencies in a specific project. An internet web site contains pointers to hundreds and even thousands of web pages. Research over the last fifteen years has tried to develop automatic methods for creating pictures of these types of applications. One of the most common methods is to create a hierarchical layout of the graph. In this case, vertices of a graph are assigned to levels so that all edges point in the same direction.

There are many published hierarchical graph drawing algorithms but they either need human intervention, work well for graphs with certain properties or run too slowly to be acceptable solutions for the general problem.

This thesis presents an algorithm for generating straight-line planar drawings of directed acyclic graphs that tries to deal with these issues. The algorithm is a modification to the one presented in [GKNV93]. It takes a description of a directed graph and assigns the nodes and edge paths to a set of Cartesian co-ordinates. It also introduces the use of flat edges into the drawing. A flat edge is defined as an edge between two nodes that exist in the same level of a hierarchy. By introducing an initial pre-processing step and the notion of flat edges, to the solution in [GKNV93], the efficiency of the algorithm is dramatically increased while layouts of equal or higher quality are produced.

1.1 Overview

This thesis is divided into seven chapters. The first is the introduction. It gives an overview of the thesis, describes the motivation, and the general ideas behind the algorithm. The second chapter reviews basic terminology used in this thesis. The third looks at previous work done in this area of directed graph drawing. The fourth describes the proposed algorithm in detail. The emphasis will be placed on describing the two heuristics used to modify the results of [GKNV93]'s work. The fifth chapter shows the results of the performance of this algorithm, while chapter six presents an analysis of these results. In particular the quality and the efficiency of the results from this algorithm will be compared with those of [STT81] and [GKNV93]. The final section discusses future extensions to the algorithm and open problems in hierarchical graph drawing.

1.2 Motivation for this algorithm

There are many ways of representing information. A graph, G, represents the relationships between objects. Objects are represented as vertices and their relationships are represented as edges. A family tree is a typical graph. People are the vertices and their parent - child relationships are the edges.

In computer science, there are many well known applications that can be modeled by graphs. Computer networks form graphs. Computer terminals and servers are considered vertices while edges represent the physical connections between them. In object-oriented programming, software engineers use graphs to show how functions, procedures and data structures are related to one another within a program. The internet is also one huge graph. Every web site is a subgraph of the entire infrastructure of the net. Each hyper-link is an edge to another web page or node. There are many other examples but the basics always remain the same. Vertices are connected via an edge to show that there is some relation between them.

In order to deal with large graphs and the information they convey, human beings prefer to visualize them. The layout of any given graph can assist with this visualization. In particular, by laying out a graph in a specific way, many interesting properties may be revealed. However, finding these properties and visualizing them readily is a very difficult problem. Because graphs in their general form have no geometric properties, there are an infinite number of ways that they can be drawn. Thus in the past twenty years there has been a significant amount of research done on developing algorithms which are efficient and produce "good" layouts.

A good layout depends on the aesthetics of a drawing. Aesthetics are geometric properties that are applied to the graph in order to create a visual representation. Aesthetics include constraints on the length and shape of edges, the minimal and maximum distance between vertices, the shape of vertices and the overall topology of the graph. How well an algorithm produces a layout with respect to these aesthetic criteria is a measure of how "good" the layout is.

Unfortunately this may not be as straight forward as it seems. In many cases aesthetic criteria overlap. For example a layout that maximizes the amount of symmetry may not necessarily minimize the number of edge crossings. If both these criteria are desired, a compromise or a ranking of importance must be found. Furthermore maximizing or minimizing the degree of certain criteria may be computationally intractable. It has been proven then finding maximum subgraphs with symmetry is NP-complete. In general this entire problem becomes subjective. It is the user who ultimately defines what characteristics are important to be shown in a layout. This has led to many algorithms designed specifically for special cases of graphs. This thesis looks at the general problem of drawing directed graphs.

Directed graphs have a very important property. Edges have a direction. Ensuring that edges point in the same general direction is an important aesthetic criterion when drawing digraphs. This has led to the development of hierarchical drawing algorithms. The idea of a hierarchical drawing is to orient edges so that they point in the same direction, in order to create a hierarchy of information. Obviously not all directed graphs can be drawn perfectly in this fashion but it does form a simple model

Another important aesthetic criterion is avoiding visual anomalies that do not convey information about the graph. This usually takes the form of minimizing the number of edge crossings and minimizing the number of bends in edges. Edge crossings clutter the graph and make it difficult to see which vertices are connected. A third criterion is to keep edges short. This improves the readability of a graph. Edges that are short make it easy to see which vertices are adjacent to each other. Another criterion is based on the idea of balance. If a graph has its vertices evenly distributed throughout the picture, a drawing is considered more organized and well spaced. This results in a picture that is easier to read. Most algorithms in the literature do not take this in to account. For example [STT81] does not deal with it at all while it takes on a small role in [GKNV93].

The final aesthetic criterion deals with keeping a picture within the bounds of whatever physical media is being used. For example a layout is not practical if it cannot appear fully on the screen. Therefore minimizing the overall area of the layout is important. This is rarely addressed in the current literature; however, it becomes quite critical when drawing large graphs.

A further issue which is not concerned with aesthetics, is efficiency. There are few algorithms that produce decent quality hierarchical layouts for large directed graphs in a practical amount of time. Large, in this case, means a graph of a few hundred vertices and edges. The reader will see that a graph composed of 75 vertices and 157 edges takes nearly 12 hours to calculate a layout using the algorithm present in [GKNV93] on an IBM RS6000 computer. This cost is unacceptable especially for a graph that is so small.

The algorithm described in this thesis attempts to deal with this issue as well as all of the aesthetic issues described above. Its goal is to efficiently produce well balanced hierarchical layouts that minimize the overall area.

The following sections gives a general overview of how this algorithm accomplishes this task.

1.3 Proposed Algorithm

This thesis presents a graph drawing algorithm that is a modification of the result presented in [GKNV93]. The idea is to improve on the quality and efficiency of the final layout.

The algorithm consists of a four stage process as seen in Figure 1. In the first stage, a pre-processing step is used to apply geometric properties to the graph. These constraints are then used in solving the hierarchical rank assignment problem in stage two. This stage finds an optimal solution to minimizing the overall edge length of the graph within the constraints imposed by stage 1. The third stage reduces the number of edge crossings by rearranging the order of vertices in each rank. The algorithm sweeps up and down the ranks using heuristics called the median and transpose methods to "smartly" reorder the position of vertices within each rank. This continues until a global optimum is reached. The fourth stage then fine tunes the drawing by performing local optimization at every node. It assigns (x,y) co-ordinates so that a compromise between the minimization of the overall edge length (Euclidean distance) and the "straightness" of edges is found. The algorithm then outputs the completed drawing to a file or to a graphical front end which displays it.

Figure 1 - Main Algorithm

```
1. Pre-process();
2. Ranking();
3. Ordering();
4. Positioning();
```

The major contributions to this algorithm are the pre-processing stage and some modifications to the third and fourth stages. Unlike the work done in [GKNV93], not all edges have a minimum length of 1. The pre-processing step partitions the edge set into three subsets. These sets have a minimum length configuration of 0, 1 or 2 respectively. This results in some edges connecting vertices that have the same assigned rank. These edges are called flat edges. The algorithm then ensures that all of these flat edges are pointed in the same direction, preferably from left to right across the hierarchy. Edges that are assigned minimum lengths of 2 are used to "bump" vertices down the ranks. This reduces the amount of overlapping or crowding of vertices in full ranks.

The addition of flat edges not only improves the quality of layouts in some cases but also increases the performance of the third and fourth stages. It turns out that each flat edge in the layout reduces the overall edge length of the graph by at least one, when dealing with the final two steps. Thus by maximizing the number of flat edges, the algorithm is maximizing its efficiency. The notion of "bumping" was introduced to fix the problem of assigning too many flat edges. These two ideas will be demonstrated later on in this thesis.

Chapter 2 - Review of Terminology

This section reviews some of the terminology used throughout this paper. The first part defines many terms that come from graph theory. The second part introduces some new references that are specific to this paper and algorithm.

2.1 Graph Terminology

A **directed graph**, G, is defined as a pair (V,E) where V represents a finite set of **vertices** (sometimes referred to as **nodes**), and E is a finite set of ordered subsets (u,v) where $u,v \in V$. If E is the **edge set** then an element of E is an **edge**. Edges may have a defined value based on their importance with respect to the rest of the edges. This value is called the **weight**. A **loop** is an edge (u,v) where u = v.

A vertex v is **adjacent** to a vertex u if there is an edge (u,v), or (v,u).. We say that (u,v) is **incident from** or **leaves** vertex u and is **incident to** or **enters** vertex v. Vertex u is also referred to as the **parent** of **child** v. This means that a vertex in a directed graph also has an **in-degree** and an **out-degree**. The in-degree of a vertex is the number of edges incident to it. The out-degree of a vertex is the number of edges incident from it. The **degree** of a vertex is the sum of its in-degree and out-degree. A **source** vertex is a vertex with in-degree 0, while a **sink** node is a vertex of out-degree 0. A **neighbor**, v, of a vertex u, in a directed graph is any vertex that has an edge (u,v) or (v,u).

A **path** of **length k** from a vertex u to a vertex z is a sequence $\langle v_0, v_1, v_2, \ldots, v_{k-1}, v_k \rangle$ of distinct vertices, except for v_0 and v_k where $v_0 = u$ and $v_k = z$ and $(v_i, v_{i+1}) \in E$ for i = 1, 2, ... , k - 1. A path **contains** vertices $v_0, v_1, \ldots, v_k$. A path forms a **cycle** if $v_o = v_k$ and the path contains at least one edge. A graph with no cycles is said to be **acyclic**.

A graph is **connected** if every pair of vertices is connected by a path. An **undirected graph** has no order placed on the edges. An undirected connected acyclic graph is said to be a **tree**. A unique distinguishing vertex of a tree is called its **root**. For

every finite undirected connected graph, G, there is a **spanning tree**. A spanning tree is a tree that includes all the vertices of G and uses only edges of G.

A connected directed graph is said to be **hierarchical** if it is acyclic. A vertex u, is an **ancestor** of v, if there is a path from u to v in a directed graph. In this instance, v is a **descendent** of u. For further information on graph terminology please refer to [HAR69].

2.2 Graph Drawing Terminology

A **drawing** of a graph is an assignment of the vertices and edges of the graph to a set of Cartesian co-ordinates. Vertices are then drawn at their (x,y) position and edges are drawn using straight-lines along their projected (x,y) paths.

A **hierarchical drawing** is a drawing that assigns each vertex to a **rank**. A rank or, equivalently a level, is a line in the plane that runs parallel to all other ranks. A **hierarchy** is the composition of all these ranks. When the vertices are actually placed on the screen, all vertices in the same level will be placed in a line, parallel and in between the vertices on the level above and the level below. For the purpose of this algorithm, the **distance** between two vertices u and v, is defined as the absolute value of the difference between the rank of u and the rank of v. For notation purposes the rank of v is defined by the function $\lambda(v)$. The **length** of an edge (u,v) is simply the distance between u and v. The main goal of this algorithm is to minimize the overall edge length without having too many vertices in any particular rank. Thus this definition is very important in understanding this algorithm. A hierarchy is said to be **proper** if all edges have a length of 1, and all descendants of a vertex are assigned to a rank greater than its ancestor.

One problem however, is that a directed graph may have a hierarchical layout which consists of only one rank. This results in very unsatisfactory layouts. So a minimum value constraint is given to each edge. This is called the **minimum length** denoted as $\delta(e)$ where e is an edge. It can be any integral value greater than or equal to 0. In our final layout, all edges will have length greater than or equal to there minimum length constraint.

Another important characteristic of this graph drawing algorithm is that of edge crossings. Two edges intersect or **cross** if the line segments representing the edges intersect. Trying to minimize the number of edge crossings is highly desirable in a good layout.

The **layout** of a graph is the placement of vertices and edges in order to create a physical representation. In terms of the **layout** or equivalently picture of a graph, a few items need to be defined. This thesis specifically deals with heirarchical layouts, so the **height** of a layout is defined as the total number of ranks. The **order** of a rank is the number of vertices in it. A layout is said to have a **width** equal to the highest order in the hierarchy. These terms will be used in discussing the properties of a layout.

Chapter 3 - Literature Review

Over the last twenty years, there have been many attempts to develop an algorithm which provides good drawings for any general graph. In most cases, they work well for some graphs and not for others. Some typical problems for any algorithm include unsatisfactory layouts for certain graph configurations and computationally expensive for large graphs. Many papers have been written to address these issues; however, no one has come up with a solution that is significantly better than any of the others.

The same is true of drawing directed graphs. Since 1977 when Warfield developed his "Crossing Theory and Hierarchy Mapping" in [WARF77] most research papers have tried to follow his basic idea. First assign vertices to their proper levels in the hierarchy and then try to minimize the number of edge crossings while still maintaining this hierarchy. In [STT81], Sugiyama, Tagawa, and Toda (from now on referred to as STT) introduced an additional aspect to the layouts. They suggested that long edges (i.e. edges between vertices separated by > 1 level apart) should be as straight as possible. That resulted in the addition of a third step to Warfield's original model. They also introduced a new heuristic for the removal of edge crossings. This new heuristic turned out to be the cornerstone for many research papers focused on reducing the number of edge crossings in a hierarchical drawing. Also at this time, Carpano suggested a similar approach to removing edge crossings [CARP80]. His method was based on an iterative solution. His heuristic continued to run until a global minimum was reached.

Later, in 1986, Eades and Kelly showed that minimizing the number of edge crossings in a 2 - level system is NP-complete [EADES86]. This proved that the use of heuristics for removing edge crossings was a "good" way of solving this problem. It also seemed to reduce the emphasis on developing good heuristics for the removal of edge crossings and led to a broader view of graph drawing.

At this time Battista, and Tamassia [BT88] presented an O(N log N) algorithm for drawing planar directed acyclic graphs. Its limitation, however, is that it only works for st-graphs. An st-graph is an acyclic digraph that has one source node, s, and one sink node, t,

and there exists an edge (s,t). In 1994 Bertolazzi, Battista, Liotta, Mannino [BBLM94] extended this, to test whether a 3-regular acyclic digraph has an upward planar drawing in $O(n + r^2)$ time where n is the number of vertices, and r is the number of source and sink nodes. An upward planar drawing is simply a hierarchical drawing with no edge crossings. These results show that for specific types of graphs, there exist fast algorithms that produce good layouts, but for large general directed graphs there have yet to be any significant improvements.

In the late 80's, algorithms for drawing small graphs with good layouts were quite common. So researchers looked to the problem of drawing large graphs from a small graph point of view. The basic thought was that by breaking up the graph into smaller parts, large graphs could be drawn by finding good layouts for the small graphs, and then incorporating these small layouts into one large one. In 1991, Messinger, Rowe and Henry presented a divide and conquer approach in [MRH91] based on this idea. Their results however showed that the divide and conquer approach to graph drawing is not a good one. The main reason for this is the quality of the final layout. Individual subgraphs may be well drawn, but when incorporated with the rest of the graph, the picture looks awkward and messy. This can been clearly seen in instances where there are numerous edges connecting each subgraph. The idea of grouping sets of vertices together comes from the work of Gestalt [GEST47]. His theories on grouping in terms of perception have lead to many algorithms including a rule-based approach to layouts [KOS94], and a recursive clustering method [NORTH93]. The problem with both of these solutions is that deciding which set of vertices constitute a subgraph, is extremely difficult. In both of these algorithms, the assumption is made that these subgraphs are determined manually or by some external device.

North's basic algorithm came from a paper he co-authored in the late 80's [GKN88]. In it, an algorithm was presented that produces layouts where the assignment of levels in a hierarchy is optimized so that the overall edge length in the graph was minimized. Then in 1993 he co-authored another paper based on the earlier one [GKNV93]. Here, the initial algorithm was extended so that curves were used to represent edges and the "straightness" of the edges was optimized.

Another approach is to create a database of drawing algorithms, and then to automatically choose which algorithm would be best suited to draw a particular graph. This was done in [BBL95]. The one problem with this method however, is the amount of human intervention needed to decide the best suited algorithm to produce the best picture.

There are many digraph drawing algorithms but most of them follow the same basic strategies that are shown above. The ideas presented in this paper are based on the algorithms contained in [GKNV93]. For a more in depth review, the reader is refer to the annotated bibliography in [BETT94].

The next section looks specifically at 2 different approaches to drawing directed graphs. The first will discuss STT's work and how it affected many papers that followed. The second is a look at Gansner *et al.*'s "A Technique for Drawing Directed Graphs" [GKNV93] and the limitations that it presents.

3.1 Sugiyama, Tagawa and Toda

Most recent algorithms that draw directed graphs in a hierarchical way are based on the ideas that were proposed in their paper in 1981 [STT81]. Figure 2 shows their 4 step process in which a description of a directed graph was input and a picture of the graph was the resulting output. The aesthetic goals of this algorithm are to orient edges in the same direction, minimize the number of edge crossings, and to keep edges short and straight.

Figure 2

1. Transform into hierarchy.
2. Reduce edge crossings.
3. Horizontal Positioning of vertices.
4. Display on terminal.

The first step was to transform the graph into a proper hierarchy. STT do not go into details about how they do this but it is inferred that all cycles are eliminated by reversing the direction of some of the edges and all edges that span more than 1 level are replaced by a series of dummy vertices and edges. This ensures that the graph is a proper hierarchy.

The second step was to reduce the number of edge crossings in the graph. Once the hierarchy was found, an ordering was imposed on the vertices in each level. Therefore a vertex in level i was assigned to some position with respect to the other vertices in the same level. By modifying the order for each level, the number of edge crossings in the graph can be reduced in most cases. The heuristic they used was called the *barycentric method*. The basic idea is as follows. Starting at level 2 and continuing to level k (k being the maximum level) each vertex in a level is assigned a position based on their ancestors in the level above. This position is calculated as the average of the positions of a vertex's ancestors in the previous rank. They are then sorted based on their new position values. Once this has been completed for all levels the number of edge crossings is calculated for the entire graph and this is compared with the previous value. The algorithm sweeps up and down the levels until there is no more improvement in the number of edge crossings.

Once this has been completed, the problem of assigning actual (x,y) co-ordinates to the vertices is solved using a heuristic called the Priority Layout Method. The basic idea is to make edges as straight as possible. Therefore long edges which are composed of dummy vertices will be relatively straight and easier to read. The process is similar to the barycentric method discussed above. By starting at level 2 and iterating through until level k, each vertex in each level is assigned an integral x co-ordinate. These co-ordinates will maintain the ordering imposed by step 2. Which vertex is the next to be assigned an x co-ordinate is based on its priority number. Dummy vertices are given higher priority than non dummy vertices because they compose long edges. Because dummy vertices have an in-degree of 1 they can be placed directly below their ancestor. All other vertices are assigned x co-ordinates based on the integral average of their ancestors. If these values break the constraints of the imposed ordering then they are inserted as close as possible without breaking any of these constraints.

The final step reverses all previously reversed edges, removes dummy nodes, and outputs the final layout onto the screen.

This algorithm is fast and works quite well, however there are many areas that can be improved upon. First of all, the problem of partitioning the set of vertices into levels is never explicitly addressed. In fact very little work appears in the literature about this

problem. Most research tends to deal with the edge crossing problem more than setting up the organization of the hierarchy. One of the limitations that this algorithm has is that edges may be straight but sometimes unnecessarily long. This results in more dummy nodes than needed and therefore the computational time is greater. Furthermore, the edges sometimes have sharp bends as they leave or enter nodes. This is undesirable because it makes reading the graph more difficult. Another observation is that in most drawings the perimeter is unnecessarily large. This means that the width/height ratio becomes extreme (i.e. one of the width or height is much greater than the other), and screen limitations start to interfere with the overall quality of the layout. This becomes a factor when scaling is needed. Any layout can be scaled down to fit on the screen, but to ensure the layout is unchanged, the scaling must be done by a constant factor. Therefore if either the height or width is much greater than the other, the final layout will be scaled down much more than necessary. Individual vertices may start to overlap one another or entire levels will become scrunched together, thus making reading the graph more difficult than necessary.

The next section looks at another algorithm that is similar in design but specifically looks at solving the level assignment problem and (x,y) assignment problem.

3.2 Gansner, Koutsofios, North and Vo

Like STT's algorithm, Gansner *et al.* propose a 4 step algorithm that draws a directed graph in a hierarchical layout as seen in Figure 3. This algorithm differs in two distinct ways from the STT algorithm. It explicitly solves the level assignment problem such that the overall edge length of the graph is minimal and all edges have similar orientation. It also maximizes the "straightness" of all edges within the graph. Moreover, because their implementation uses curves to represent edges, their algorithm tries to avoid sharp bends as much as possible.

Figure 3

```
1. Rank()
2. Ordering()
3. Position();
4. make_splines();
```

The first step ensures that the graph is acyclic by reversing a subset of edges. Then the hierarchy is found by solving the following mathematical formulation: minimize the

overall length such that an ancestor of a vertex always appears in a lower level than its descendent. They solve it using a network simplex approach. Even though this type of solution is not proven to be polynomial, in practice it requires very few iterations even on graphs composed of 250 vertices.

The second step is similar to STT's. By replacing edges that separate vertices more than one level apart with a connected chain of "virtual" or dummy vertices, an ordering can be imposed on each level. By manipulating this order, the number of edge crossings can be decreased. Gansner *et al.* propose two modifications on existing heuristics called the median and transpose methods. They are claimed to work 20-50% better than the barycentric method.

The third step is to assign (x,y) co-ordinates to each vertex. By applying the same algorithm as in step 1 to a specially transformed graph, an optimal solution can be found to the problems of keeping edges straight, minimizing edge length and avoiding sharp bends. The problem however is that this transformation results in a much larger graph, and the computational time becomes much more expensive.

The final step computes the curves that will represent the edges. For each edge a set of control points are calculated such that the curve they represent is as smooth as possible and yet avoids all obstacles like other nodes and edges. The control points will then be used to draw piecewise Bezier curves. The results of using curves instead of straight-lines are aesthetically pleasing but does not improve on the overall readability of the graph.

This algorithm is a drastic improvement on many of the areas that STT's algorithm lacked. It does have its problems as well. In some cases, it is sometimes desirable to have longer edges. When there are a large number of vertices in a single level, it is better to spread them out among more than one level. This is particularly beneficial if one of the adjacent levels has few vertices. Also, the third step can become quite slow depending on the size of the graph. Thus, even though this algorithm is optimal with respect to minimizing the total edge length of the graph, it still does not necessarily produce a "perfect" drawing. In most cases, a drawing that minimizes the total area of the graph is better than one that minimizes the total edge length.

Because the algorithm described in this paper uses the algorithm presented in [GKNV93] explicitly, the following sections will describe it in detail.

3.2.1 - Breaking Cycles

By definition a hierarchy has no cycles. A directed graphs, however, can have cycles. So before vertices can be assigned ranks properly, all cycles must be eliminated from the graph. To deal with this problem, some pre-processing is done that detects cycles and breaks them by reversing certain edges. It should be noted that this is done internally and that when drawn, these edges will have their original direction. There are a number of ways that this can be accomplished but it turns out that the simplest way is sufficient. By doing a depth-first search starting from a source node, if it exists, a partial order is defined by the depth-first search tree. The edges are then partitioned into two sets: tree edges and non-tree edges. By incrementally looking at all non-tree edges and adding them to the initial partial order, each non-tree edge can be placed into one of three categories: cross edges, forward edges or back edges. Cross edges connect unrelated nodes in the partial order while forward edges connect an ancestor to a descendant. These types of edges when added to the partial order do not create cycles. Back edges, or edges that connect a descendent to its ancestors, do create cycles. By reversing a back edge, one creates a forward edge. So by reversing every back edge, all cycles are broken.

What is important to note is that in some cases by breaking cycles in this manner, a large number of edges may be reversed depending on the graph input. Hence, when the actual drawing is performed there will be a large number of edges pointing in the opposite direction to the rest of the edges in the graph. One way of dealing with this is by reversing the minimal set of edges in the graph so that no cycles occur. This however, is the same as solving the feedback arc set problem which is NP-complete [EADES86]. So solving this problem in this manner is computationally unacceptable. It turns out that the depth-first solution in most cases works effectively as well as quickly.

3.2.2 - Rank Assignment

As stated earlier, the ranking part of the algorithm tries to minimize the sum of the over all edge length in the digraph while maintaining a hierarchy. Recall that the length of an edge (v,w) is $\lambda(w) - \lambda(v)$ where $\lambda(w)$ equals the rank of vertex w. Then the hierarchical property of the graph can be enforced by having all edge lengths greater than the minimum length constraint $\delta(v,w)$. Therefore the following is a mathematical formulation of minimizing the sum of the overall edge length for a hierarchical layout.

Integer Program 1

$$\min \sum_{(v,w)\in E} \omega(v,w)(\lambda(w)-\lambda(v))$$

$$\text{subject to: } \lambda(w)-\lambda(v) \geq \delta(v,w) \forall (v,w) \in E$$

The weight ω of an edge (v,w) is used to give a higher priority to special edges. In most cases the weight of an edge is equal to 1. δ is the minimum length function and is used as a vertical positioning constraint. It ensures that the hierarchical property of the layout is maintained, when $\delta(v,w) > 0$.

Now, we have a linear objective function that is subject to a system of linear constraints, that is, a linear programming problem. There are known ways of finding an optimal solution to this problem in polynomial time; however, as in [GKNV93], a good method of solving it is called a network simplex. This has not been proven to be polynomial but for this problem tends to work using very few iterations. A brief description can be seen in Figure 4.

Figure 4

```
1. init_rank();
2. feasible_tree();
3. while (e is an edge with cutvalue < 0)
4.       f = replacement edge for e;
5.       exchange(e,f);
6. end
7. normalize();
8. balance();
```

The basic theory behind a network simplex solution is to find an initial feasible solution, and then try to improve on it by increasing one variable at a time. A solution is

feasible if it lies within all constraints imposed in the initial problem. In terms of minimizing the sum of the overall edge length, all edge lengths must be greater than or equal to their minimum length constraint. How far a solution is from the optimal is measured by a series of slack variables. Note that the slack variables in this case are defined as the length of an edge minus its minimum length constraint. An edge is considered tight if its slack is 0. As the slack variables decrease in value because of the increase in the other variables, the closer the feasible solution becomes to the optimal one. However, when an increase in any of the variables will make the solution infeasible, the solution is considered optimal. As a rule, the choice of which variable to be increased in each iteration is based on what variable will move the solution closest to an optimal solution. It is then increased as much as possible so that its new value will still induce a feasible solution. This is continued until the optimal solution is reached. This is obviously a simplified explanation but it will suffice for the purpose of describing how the ranking assignment works. For more information on this type of solution the reader is referred to [HEEST83].

So how can this type of solution be applied to this graph problem? It is in fact very simple.

The first step (lines 1-2) of Figure 4, is to construct an initial feasible solution. This is done as follows.

Figure 5

```
1. while (V ≠{}) do
2.     for all v ∈ V with no "in" edges
3.         AssignRank(v);
4.         delete v and all its edges.
5.     end for
6. end while
```

In other words, each source vertex is assigned a rank then deleted from the graph (lines 2-5 of Figure 5). A vertex *v* is assigned a rank based on the following formula.

Formula 1

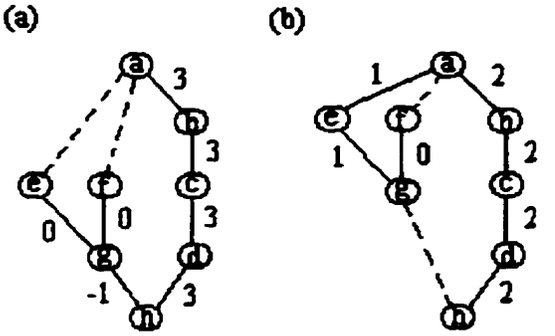
$$\max(\lambda(u)+\delta(u,v)):\forall(u,v)\in E$$

This way v is assigned the minimum rank that will keep the solution feasible. At the same time it is also easy to construct an undirected spanning tree of the graph G. This underlying undirected spanning tree will have important implications for the network simplex algorithm. The edge lengths of the spanning tree constitute the variables that the network simplex algorithm will solve for, in order to reduce the overall edge length. The slack variables will be defined as the length of an edge, e, minus δ(e). Therefore if the slack(e) is 0 then that edge length for e is optimal and the edge is said to be tight.

Now the question is how can one find an optimal feasible ranking (lines 3-6 of Figure 4). Gansner *et al.* discovered the following geometric property. In most cases, by removing a special edge from the spanning tree and replacing it with a non-tree edge, a new induced ranking can be constructed which has a reduced overall edge length. This special tree edge has the property that, by increasing the length of it, one or more non-tree edges are being shortened by the same amount. Thus the overall edge length was being reduced. In order to capture this property quantitatively, the authors used the notion of a cutvalue. A cutvalue is assigned to every edge of the spanning tree. By deleting an edge from the spanning tree, it is broken into two components: the head and the tail. The head being the part of spanning tree that was connected to the "head" of the edge, and the tail being the remaining part of the spanning tree. The cutvalue of that edge is defined as the sum of the weights of all edges from the tail component to the head component including the deleted edge, minus the sum of the weights of all edges from the head component to the tail component. It turns out that a negative cutvalue implies that by lengthening that edge, more edges will be shortened than lengthened and that the overall edge length will be reduced. Therefore, if on each iteration an edge in the spanning tree with a negative cutvalue is removed from the tree and replaced by some non-tree edge that has the minimum slack value and reconnects the spanning tree, a new feasible spanning tree is found. This new solution will have a shorter overall edge length than the previous one. This continues until there are no more edges with a negative cutvalue in the spanning tree.

The resulting assignment is optimal. An example of how this works can be seen in Figure 6, adopted from [GKNV93]. All edges are assumed to be directed downward, and all edges have weight 1 and $\delta(e)$ equal to 1.

Figure 6



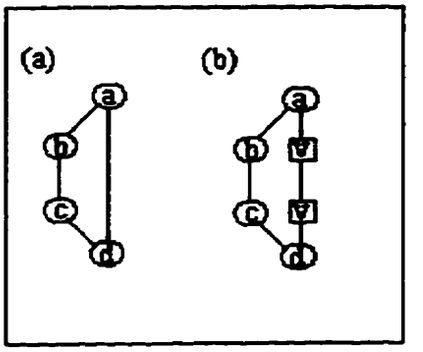
The spanning tree is composed of the non-broken lines while the broken lines are the non tree edges. The cutvalues of each is the number shown beside them. In (a) there is an edge (g,h) with a negative cutvalue of -1. By replacing this edge in the spanning tree with (a,e) and re-calculating all the cutvalues, the result is (b). Notice there are no more negative cutvalues. This implies that an optimal solution has been reached.

The final two lines (lines 7-8) of the ranking algorithm are house cleaning. Normalize() ensures that the initial rank begins at zero, while balance looks for vertices with equal in and out edge weights and multiple feasible ranks. If any exist, they are placed on the feasible rank with the fewest vertices.

3.2.3 - Minimizing the Number of Edge Crossings

Now, like most hierarchical, digraph layout algorithms, an attempt is made to reduce the number of edge crossings in the layout. The solution used is similar to the ones suggested by Warfield [WARF77] and Sugiyama *et al.* [STT81] and modified by Gansner *et al.* [GKNV93]. "Virtual" nodes are first inserted into the graph. The purpose of this it to obtain a proper hierarchical layout. For each pair of vertices connected by an edge whose length is greater than 1, this edge is replaced by a chain of unit length edges between virtual nodes. Thus the graph G is transformed into a new graph G' such that all edges connect nodes only on adjacent ranks. For an example of this the reader is referred to Figure 7. The edge between vertex 1 and vertex 4 has been replaced by three other edges of length 1, and 2 virtual nodes. This results in all edges in the new graph being of length one.

Figure 7



Gansner *et al.* use the algorithm shown in Figure 8 on this new graph to reduce the number of edge crossings. The idea is to find an initial ordering for each rank, and then try to improve the solution through a sequence of iterations. These iterations are done by traversing each rank from the first to the last or vice-versa. In the process each vertex is assigned a weight based on the relative positions of its incident vertices on the preceding rank. This weight should not be confused with the weight $\omega(u,v)$ defined in Integer

Figure 8

```
1.  order = init_order();
2.  best = order;
3.  for i = 0 to Max_Iterations do
4.      wmedian(order,i);
5.      transpose(order);
6.      if crossings(order) <
    crossings(best)
7.      then best = order;
8.  end for
9.  return best;
```

Program 1. A weight is assigned to each vertex, not to any of the edges. A new ordering for each rank is found by sorting the vertices by their weight.

The initial ordering (line 1) is done using a simple breadth-first search. By starting with vertices of minimum rank, vertices are placed in their ranks in left to right order as the search progresses. A depth-first approach could also be used in this manner. This way trees will have no initial edge crossings by construction.

To deal with calculating the weights for each vertex (line 4), a variation of the median heuristic is used. The median method was first developed by Eades and Wormald in 1986 [EADES86]. The basic idea is create a list of the positions of all incident vertices on the following rank, and to take the median of this list as its weight. In the case that there are two medians (i.e. the length of the list is even), the value of the median that places the vertex closer to the side where its children are more densely packed is used. The hope is that by placing a vertex near to more of its children, the number of edge crossings will be fewer. In fact, it was proven that the number of edge crossings will be no more than 3 times the minimum number of crossings in a two level layout by using this heuristic. However, this upper bound is still fairly high, so like in [GKNV93], an additional heuristic is used.

The transposition heuristic (line 5) switches neighboring vertices within a rank, and checks to see if there has been an improvement in the number of crossings. If there has been, another two vertices are switched. This continues until there is no more improvement. By starting at the lowest rank and working downwards, Gansner *et al.* claim that there is an additional reduction in the number of edge crossings by 20-50%. The reason for this, is that the median method only gives an approximation for the best spot to place a vertex in the ordering. The transpose method then checks to see if positions around the given one may be better in order to reduce the overall number of edge crossings.

3.2.4 - Positioning of Vertices

Now that we have a ranking and an order, the problem is to assign actual (x,y) co-ordinates so that a picture can be drawn. The premise behind the ranking and the ordering is to give constraints to the layout. The rank of a vertex defines which vertices will be lined up vertically, while the ordering defines a vertex's horizontal neighbors to the immediate left and right.

Assigning the y co-ordinate is straight forward. By giving the same y co-ordinate to all vertices of the same rank, every vertex is lined up horizontally with all the vertices in its rank. How far each rank is separated can be done in many ways. A straight forward method is to space each rank by a given constant.

Assigning the x co-ordinates can be a much more difficult problem. This layout algorithm tries to assign the x co-ordinates such that short, straight edges are preferable to long, crooked ones. By introducing a weight $\Omega(e)$, an internal value distinct from the input edge weight $\omega(e)$, long edges can be given a greater priority for straightness over shorter edges. Using this Gansner *et al.* propose the following as a mathematical formulation of the problem where xsize(v) is the physical size of vertex v in the x-direction and nodesep(G) is a previously defined constant that signifies the minimum space between vertices in the same rank:

Integer Program 2

$$\min \sum_{e=(v,w)} \Omega(e)\omega(e)\left|x_w - x_v\right|$$

subject to: $x_b - x_a \geq \rho(a,b)$

where a is the left neighbor of b on the same rank and

$$\rho(a,b) = \frac{xsize(a) + xsize(b)}{2} + nodesep(G)$$

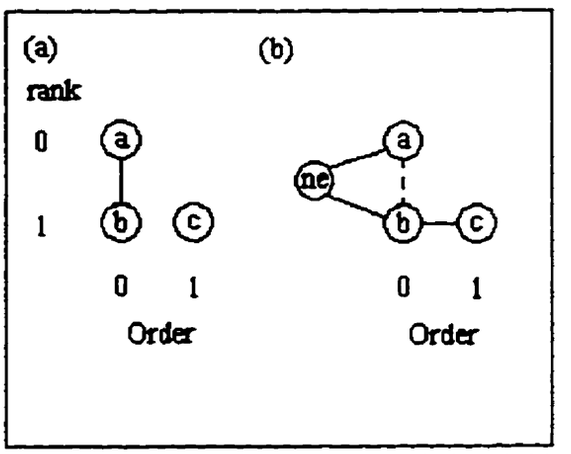
The idea of this minimization problem is to assign x co-ordinates accordingly so that edges between adjacent vertices are as short and straight as possible. The function ρ(a,b) ensures that when the vertex is actually drawn, there are no overlaps between vertices a and b.

Having adjacent vertices aligned vertically results in shorter edges. This alignment corresponds to reducing the horizontal distance between adjacent nodes. It is particularly important to have virtual nodes aligned vertically so that edges are straight. If this is not done, the edges become "spaghetti-like." This is the main purpose for the constraint $\Omega(e)$. By setting $\Omega(e)$ equal to 1 for edges between two real nodes, equal to 2 for edges between a real node and a virtual node, and equal to 8 for edges between two virtual nodes, it becomes more important to straighten long edges over short ones.

The above linear programming problem is analogous to the rank assignment problem. The idea is that x co-ordinates can be considered as "ranks". By replacing every edge $e = (u,v)$ in G by a vertex n_e, and two edges (n_e,u) and (n_e,v) a new auxiliary graph is created. The minimum length constraint δ for these edges is 0 and they have a weight, $\omega = \omega(e)\Omega(e)$. The addition of these new nodes and edges allow for the x assignment problem to be thought of as the ranking problem.

Before this works completely however, one more constraint must be dealt with. As specified, running the network simplex algorithm on the new auxiliary graph would produce a layout where all edges are perfectly straight, but all vertices would be drawn on the same x co-ordinate. Gansner *et al.* propose a simple solution. For each pair of neighboring vertices within a rank, an edge is placed between them. If b is a left neighbor of c in a rank, then a new edge (b,c) is placed such that the weight $\omega = 0$ and the minimum length constraint $\delta = \rho(a,b)$.This edge forces the nodes to be sufficiently separated but does not affect the cost of the layout. Figure 6 is a simple example of how a graph , G (seen in (a)), is transformed into a new graph G', shown in (b). Mathematically this transformation can be expressed as follows.

Figure 9



Let $G' = (V \cup V', E1 \cup E2)$. The vertex set V' contains a new vertex ne for every edge $e \in G$. The edge sets E1 and E2 are given by E1 = {(ne,u),(ne,v) where $(u,v) \in G$} and E2 = {(v,w) where v is the left neighbor of w in the same rank} and all have weight $\omega(u,v)$. Now we can express Integer Program 2 in terms of G'.

Integer Program 3

$$\min \sum_{e=(v,w)\in G'} \omega(v,w)(\lambda(w)-\lambda(v))$$

subject to: $(\lambda(w)-(\lambda(v)) \geq \delta(v,w) \forall (v,w) \in E_1$

where $\omega(v,w) = 0$.

and $\delta(v,w) = \rho(v,w)$ if $(v,w) \in E_2$

The network simplex algorithm described in Section 3.2.2 can now be used to solve the level assignment problem shown in Integer Program 3. It should be noted that the optimal solution to the level assignment on G' is also the optimal solution to the positioning problem on G, as shown in [GKNV93].

Proof:

A solution to the positioning problem on G corresponds to a solution of the level assignment problem on G' with the same cost. This can be seen by assigning each vertex n_e the x-coordinate $x_{ne} = \min(x_u, x_v)$ where u and v are vertices connected by the edge e, and x_u and x_v are the x co-ordinates of u and v. Then the cost of edges (n_e,u) and (n_e,v) is $\omega(e)\Omega(e)\,(x_u - x_{ne}) + \omega(e)\Omega(e)(x_v - x_{ne}) = \omega(e)\Omega(e)|x_u - x_v|$. Therefore the cost of a solution in G is equal to the cost of a solution in G'. Conversely, any solution to the level assignment in G' induces a valid positioning in G by construction. Also, in an optimal level assignment one of $\{(n_e,u), (n_e,v)\}$ must have length 0 and the other would have length $|x_u - x_v|$. This means that the cost of the original edge (u,v) equals the sum of the cost of the two edges (n_e,u) and (n_e,v) and the solution for G' has the same cost as G. Therefore the optimality of G' implies the optimality of G. ■

We have shown how [GKNV93] do hierarchical graph drawing using integer programming problems to find optimal solutions for the ranking and positioning aspects of

their algorithm. However, these solutions are based on constraints that are not explicitly dealt with. They suggest that not all $\delta(e)$ values are equal to one, but they never given any explanation for alternative values or techniques for assigning non-unary $\delta(e)$ values. In the next section we address this issue.

CHAPTER 4 - Assigning δ-values to Edges

The major contribution of this thesis is a modification to the algorithm described in [GKNV93]. A preprocessing step is introduced to assign non-unity δ(e) values. It uses heuristics to modify the initial properties of the input graph. Where Gansner *et al.* define the minimum length constraint to be 1, the pre-processing step partitions the edge set into three subsets where the minimum length constraints are either 0, 1 or 2. Edges that have a δ(e) = 0 have the possibility of becoming "flat" in the final layout depending on the results of the rank assignment. By having certain edges in the final layout flat, the final layout can potentially have fewer ranks and reduces the sum of the overall edge length. Furthermore if all flat edges point in the same direction (e.g. from left to right) reading graphs can be easier. Figure 10 shows the algorithm presented in this paper. Break_Cycles(), Rank(); and Position() are identical to those presented in [GKNV93]. The Preprocess() and some modifications to the ordering are the contributions of this paper to the ideas of Gansner *et al.* This chapter will look at the details of the preprocessing step, and some of the modifications needed in order for the ordering step to deal with the notion of flat edges.

Figure 10

```
1. Break_Cycles();
2. Preprocess();
3. Rank();
4. Ordering();
5. Position();
```

4.1 Pre-processing

In most cases the weight of an edge is equal to 1. δ is the minimum length function and is used as a vertical positioning constraint. Because the length of edge e = (*v*,*w*) is defined as λ(*w*) - λ(*v*), the δ(*v*,*w*) function ensures that the hierarchical property is maintained, if δ(*v*,*w*) > 0. Usually the δ function for all edges is defined as 1 but in many layouts by modifying these δ values, a better layout may be achieved. By setting some δ values to 0, and others to 2, a more equal distribution of vertices occurs throughout the ranks of the graph. The question is without any initially

Figure 11

```
1. for all source nodes, u∈V
2.       MaximalLongestPath(u);
3. end for
```

known properties how does one choose which edges to assign δ values of 0,1 or 2? We propose two heuristics. One for δ(e) = 0 assignments and one for the δ(e) = 2 assignments. The goal of the latter is to clean up the results of the former. The reasons will be discussed later in this thesis.

In order to see which edges should be assigned the δ value of 0, it was observed that long chains of vertices connected by flat edges are preferable to multiple flat edges randomly placed in the drawing. Also, each vertex can have at most one flat edge entering it, and one leaving it. This way there can be no overlapping edges (edges drawn over top of one another). So that led to the assumption that for each vertex at most one entering edge and one leaving edge would be given a δ value of 0.

The algorithm is shown in Figure 11 and is described in detail as follows. By starting at some arbitrary source node, do a depth first traversal such that for each vertex *u* encountered, of all the edges (*u*,*v*), where *v* has no in-edges with δ(e) = 0, choose an edge e = (*u*,*v*) that initiates a maximal longest path from *u* and set δ(e) = 0. The additional constraint on *v* ensures that each vertex can have at most one in-edge and one out-edge. If there are no such edges that meet this requirement, then u will not be assigned any out-edges with a δ-value equal to 0. A nice feature of this algorithm is that it is simple

Figure 12

```
int MaximalLongestPath (vertex u)
1.  if ( Marked (u) AND No-InFlatEdges (u) )
2.      return LP(u);
3.  else if ( Marked (u) )
4.      return -1;
5.  end if
6.  Mark(u);
7.  if (u is a sink vertex)
8.      LP(u) = 0;
9.      return 1;
10. else
11.     best = -1;
12.     for each edge (u,v) do
13.             length = MaximalLongestPath(v);
14.             if (length > best)
15.                     best = length;
16.                     bestVertex = v;
17.             end if
18.     end for
19.     LP(u) = best;
20.     if (LP(u) ≥ 0)
21.             δ(u,bestVertex) = 0;
22.             return LP(u)+1;
23.     else
24.             return 0;
25.     end if
26. end if
```

to code. Figure 12 shows the details of the how it works. LP(u) is the length of the longest path from vertex u and No-InFlatEdges(u) is a check to guarantee that u does not have any in-edges that have $\delta(e) = 0$. Mark(u) is a means of tagging a vertex as "already been seen."

This algorithm is based on the fact that the length of the longest path from u is 1 + the length of longest path starting with its immediate children in an acyclic directed graph. The edge which connects u to v that corresponds to the longest path from v and has no in-edges with δ-value equal 0 is assigned $\delta(u,v) = 0$. If there is more than one such edge then only one is chosen and it is done on a first-come-first-serve basis. Lines 1-5 of Figure 12, check to see if this vertex has been seen previously. If it has, and it has no in-edges with $\delta(e) = 0$, then the length of its longest path is returned. In the case that it has been seen, but it already has an in-edge with $\delta(e) = 0$, the algorithm stops. Lines 7-26 calculates the length of the longest path and modifies the corresponding edge if there is one. Since each edge is scanned once and each vertex is visited once then the computational complexity of the algorithm is O(N+M) where N is the number of vertices and M is the number of edges and a linked list implementation is used to store the graph. An example of how this algorithm works can be seen in Figure 13.

Figure 13

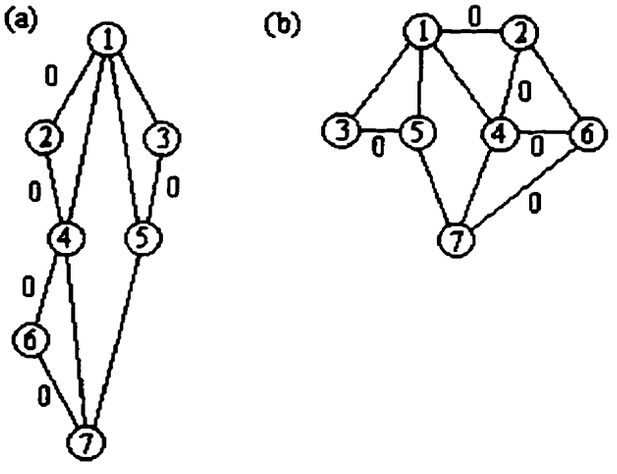


Figure 13(a) shows the layout of a graph using the ranking algorithm presented in [GKNV93] with all edges having $\delta(e) = 1$ (assuming this graph is acyclic, and all edges have direction from the lower numbered vertices to the higher ones). The '0's by some edges correspond to edges that would have had their $\delta(e)$ values changed to 0, had the graph been passed

through the preprocessing step. The result of the ranking algorithm on this altered graph can be seen in (b). Observe that the longest path is {1,2,4,6,7}. All the edges in this path have $\delta(e) = 0$. These vertices have also been all marked, leaving only vertices 3 and 5. The longest path from 3 turns out to be along edge (3,5), therefore $\delta(3,5) = 0$. Because (5,7) is the only out-edge of 5 but 7 has already been tagged, $\delta(5,7)$ maintains the default value of 1.

One problem with this strategy is that layouts can become too flat, if too many edges are assigned $\delta(e) = 0$. One way to solve this problem is by "bumping" certain vertices down a rank. This can be accomplished by assigning some edges $\delta(e) > 1$. First of all, however, it must be determined whether vertices actually need to be bumped. To do this an approximate solution to the rank assignment problem is found and then checked to see if there are too many vertices in any particular level. The approximation is calculated using the algorithm to find the initial feasible solution described in Figure 5.

This ranking is then checked from top to bottom to ensure that there are no full ranks. A rank, r, is full if the horizontal space used by the vertices in r is greater than the physical space. For example if a monitor has a resolution of 800 x 600 pixels, then a ranking that uses 1000 pixels in the horizontal direction in the final layout will not physically "fit" onto the screen. This cannot be accurately measured until after the final layout has been found. So a special calculation is used to approximate it. Recall that the xsize(u) is the length of a vertex u in the x direction and nodesep(G) is the horizontal separation between vertices. So let AverageX(r) be the average of xsize(u) $\forall\ u \in r$. Then the approximation for the total space taken up by the n vertices in rank r is:

Formula 2

$$2n(AverageX(r) + nodesep(G))$$

We double the average space taken up in each rank, because the ranks in the initial ranking is generally not as full as the final solution will be. By doubling it, we get an approximation for how full the ranks will be in the final solution.

If the value from Formula 2 is greater than MAX_X where MAX_X is some constant, then some vertices will be "bumped" down a rank. The vertices to be bumped are chosen by their out-degree. The vertex with the greatest out-degree on the previou rank (r-1) is always bumped, by assigning its shortest in-edge, e, $\delta(e) = \delta(e)+1$. If the vertex has already been bumped once, it is ignored, and the next vertex of highest out-degree is bumped instead and all approximations are re-calculated. Figure 14 shows the pseudo code for this part of the algorithm. RankFull() returns the rank first rank that does not meet the "full" criteria. ArrangeRank(r) then bumps a vertex from the previous rank and modifies the ranking. A check is done to see if the rank is now acceptable. If it is not, the process is repeated. Figure 15 shows the algorithm.

Figure 14

```
1. init_rank();
2. r = RankFull();
3. while(r!=nil) do
4.     ArrangeRank(r);
5.     init_rank();
6.     r = RankFull();
7. end
```

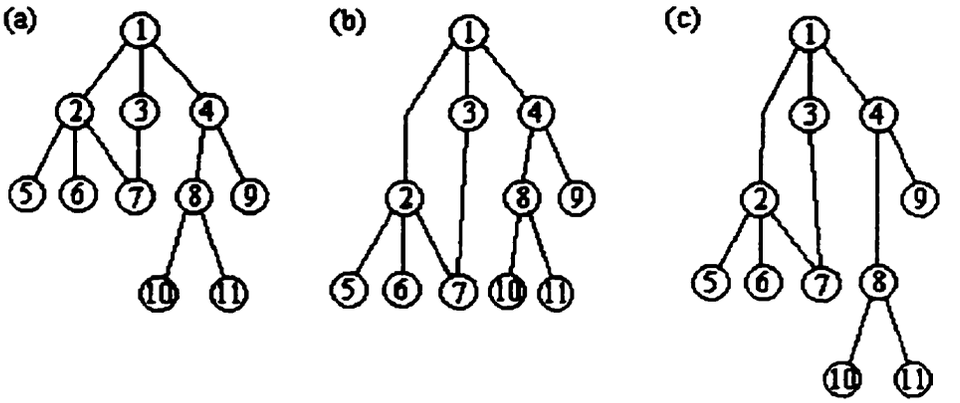
Figure 15

```
1. procedure ArrangeRank(rank r)
2. n = number of vertices in r;
3. v = GetBumpVertex(r-1);
4. while (2 (AverageX(r) + nodesep(G)) > MAX_X) AND v != nil) do
5.     Bump(v);
6.     ReRank(v);
7.     v = GetBumpVertex(r-1);
8. end
```

GetBumpVertex(r) finds the vertex that will be bumped. In this case, it is the vertex in rank r-1 with the most children. Bump(v) finds the shortest in-edge into v and adds one to its $\delta(e)$ value. Because a vertex has been moved down one rank, most of its descendants will also be forced to move down a rank. So ReRank(v) starts at v and does a breadth-first traversal down the graph modifying the ranks of vertices if necessary.

Vertices are only bumped once. There are two reasons for this. It prevents infinite loops and from an aesthetic point of view, some extremely long edges. An example of how this heuristic works and its results can be seen in Figure 16.

Figure 16



Part (a) shows a hierarchical layout of graph. Notice that rank 2 (assume ranks begin with 0 and begin near the top of the page) has 5 vertices. For this demonstration let a rank be full if there more than 4 vertices in it. Therefore rank 2 is in an unacceptable state. Part (b) shows the result after ArrangeRank() is called to try to fix this. Because vertex 2 has the greatest out-degree in rank 1 of (a), it is chosen to be bumped first. Rank 2 now has 4 vertices and is no longer full. However, rank 3 now has become unacceptable. So ArrangeRank() processes this rank. The results can be seen in (c). Vertex 8 has been moved instead of vertex 2 because vertex 2 has been bumped previously. Notice also that all ranks are now non-full and the final layout has better spacing. This demonstrates the basic idea of bumping.

The efficiency of this part of the preprocessing step is difficult to analyze. In most cases, ranks are not full. If a rank is full, choosing a vertex to be bumped requires a search of all vertices in the rank, taking O(N) time. The actual bumping of a vertex is constant, or k, but re-arranging the ranks afterward can take O(N) time as well. Therefore if each vertex must undergo this process it will take $N * (c_1N + k + c_2N) = O(N^2)$ for some constants c_1, c_2. However in most cases very few vertices need to be bumped. In fact if too many vertices (more than 1/2 of them) are bumped then the process becomes counter productive, as is shown in the results section. By adding a simple check that counts the

number of bumped vertices, the process can be stopped at any particular point, K where K is the maximum number of vertices bumped. In this case, the algorithm becomes K * O(N) = O(N).

Therefore the entire preprocessing step can take O(N+M) + O(N) time. This is much more efficient than the other stages of the algorithm presented in [GKNV93]. In fact, the time spent in computing this step is insignificant with respect to the rest of the algorithm. What is interesting however, is that by doing this, the final layout in most cases is better than those found by using just the ideas presented in [GKNV93]. Also, the preprocessing step makes a huge difference in the running time of the algorithm. These claims will be discussed in chapter 5.

4.2 Modifications to the Ordering Procedure

Recall that the Ordering() procedure in [GKNV93] uses heuristics to modify the ordering of vertices in the ranks, so that the number of edge crossings is reduced. One problem with this solution however occurs when dealing with flat edges. If vertices connected by flat edges are moved around individually, it is quite possible the edges may overlap one another, go through a vertex, or point in the opposite direction to other flat edges. To stop this from happening, a pre-processing step goes through one rank at a time looking for vertices that have edges connecting vertices in the same rank. These vertices are then placed side-by-side in the ordering. For the purpose of this paper the parent vertex is placed to the left of the child. This way all flat edges will point from left to right.

An important property of vertices connected by flat edges is that their ordering must be fixed. So any modification of the ordering must reflect this initial ordering. This is done by merging chains of vertices connected by flat edges into individual nodes. These nodes are then moved around as deemed necessary by the transpose and median methods. Once the ordering has been finalized, they are returned to their original configuration.

This is beneficial because it reduces the number of vertices that the node ordering heuristics will have to deal with. In cases where there are long chains of vertices connected by flat edges, huge improvements in efficiency can be seen. This however may result in a greater number of edge crossings.

Chapter 5 - Results

5.1 Introduction

When comparing the results of two different graph drawing algorithms, it is not enough to say one picture looks better than another. The questions of how and why it is better must be answered. The answer lies in developing a set of measurable aesthetic criteria, that can be used to compare the respective layouts. For example in the literature many algorithms quote the number of edge crossings as a measurable characteristic. Since many algorithms deal with trying to minimize the number of edge crossings, this characteristic is very important to their results. How these results compare to those of other existing algorithms is a measure of how well the algorithm worked.

The quality of the picture is not the only important feature of the algorithm. Efficiency is always an important measurable quantity. In fact the major problem in drawing directed hierarchical graphs is the computational time needed to draw large graphs. In the past ten years much of the work done has been involved in modifying well known algorithms so that they work faster and on graphs with hundreds or thousand of vertices and edges.

This section analyses the results of using flat edges as a pre-processing step to Gansner *et al.*'s hierarchical directed graph drawing algorithm called "dot". It will be shown that with the pre-processing step added to dot (which from now on will be referred to as "flatdag"), pictures of equal or greater quality will be produced in a far more efficient manner than dot.

This will be done in four sections based on the analysis techniques discussed in [JOHNS96]. The first will define the basic attributes of a graph, and how they will be measured quantitatively. The second will compare the quality of the drawings output from flatdag and dot with some originally published drawings from Sugiyama *et al.*'s paper on drawing hierarchical directed graphs. The third section will look at the quality of drawings of flatdag compared with those specifically from dot. The final section will then look at a comparison of the efficiency of these two algorithms.

5 .2 Objective Measures - Aesthetics

It is not an easy problem to objectively measure the aesthetics of a graph. However, since the final goal of flatdag is to produce layouts that are easily readable, have comparable height and width, and are computationally attractive, the following measurable characteristic will be used to compare the layouts of flatdag with other algorithms.

The first measurable attribute of any layout is the number of ranks or levels in the hierarchy and how it compares with its width. Many graph drawing algorithms produce pictures with a disproportionate number of levels compared to the width. This results in two potential problems. Because human beings perceive pictures as whole before they perceive specifics [GRAY91] and [GEST47], a good layout should have an even distribution of vertices in both the horizontal and vertical direction. Having vertices placed in such a manner provides even spacing between nodes and uniformity throughout the drawing. The second and more important problem, is that having more levels results in having a greater number of dummy or virtual nodes added to the graph. Most hierarchical drawing algorithms for directed graphs insert these nodes into their layouts in order to produce better pictures. However, this enlarges the size of the graph. Therefore reducing the number of virtual nodes will be beneficial for the efficiency of the algorithm.

The ratio of width to height could be measured in terms of the maximum x and maximum y coordinate that occur in the final layout. Unfortunately this is not the best way of doing it. Note that these values can differ greatly by the value of what constants are used to express the distance between ranks, the minimum distance between vertices in the same rank, not to mention the size of each vertex. So for this paper, the ratio will be measured by the total number of ranks (height), and the maximum number of vertices in any rank in the final layout (width). It should be noted however, that in this paper the distance between two consecutive ranks is approximately 2.5 times the minimum distance between the center of vertices in the same rank. Because a "good" ratio between width and height of a drawing in principle is 1:1, for dot and flatdag an equivalently good ratio is 2.5:1.

One way to evaluate how much the number of virtual nodes adds or detracts from a layout of graph is by measuring the sum of the total edge length. Since long edges reduce the readability of a layout, it is important to have this minimized. Recall that for the purposes of this paper the length of an edge (u,v) is not the Euclidean distance, but defined as the positive difference between rank(u) and rank(v).

Another measurable attribute that deals with the readability of a layout is the number of edge crossings. Because this tends to be a very important measurable quantity it will be used in this section as a measuring stick to compare these results with those of existing algorithms.

So for measuring the aesthetics of a layout these three qualities will be measured:

- the relationship between the width and the height
- the sum of the total edge length
- the number of edge crossings in the graph.

They will be used to compare and contrast the results of the algorithm proposed in this paper and those of previously published algorithms.

5.3 Objective Measures - Efficiency

Measuring something subjectively, like the aesthetic qualities of a drawing, is not any easy task, but the opposite is true for something objective. The efficiency of an algorithm can be determined in a number of ways. The first is to measure how many seconds it takes to complete. The second is to monitor how many iterations are undergone before the algorithm is complete. Because the implementations of flatdag and dot are not necessarily optimized, the number of iterations undergone may be more significant than the actual speed of the algorithm. It should be noted however, that the only differences in the implementations is the pre-processing step that is used in flatdag. Thus the difference in efficiency is a reflection of what the pre-processing step does to the initial characteristics of the graph and not the different implementations.

Another method of measuring the efficiency is by following how the graph changes as it goes through each step. For example Gansner *et al.* noted that the positioning step uses a disproportionate amount of time to complete compared with the other stages. This is caused by the increased size of the augmented graph used in this stage. By knowing the number of edges and vertices that are used in the positioning step, it is easy to see changes as a result of which algorithm is used.

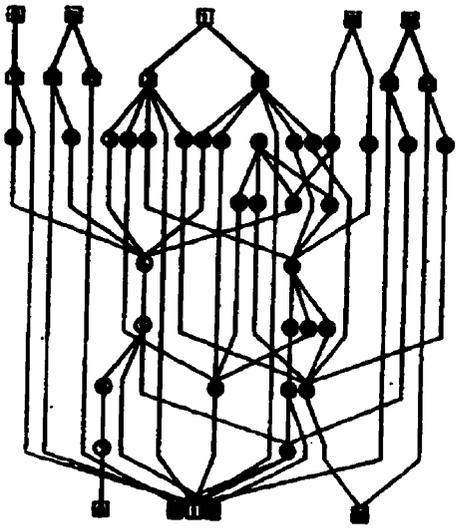
Thus the efficiency of flatdag and dot will be measured in three ways: actual time taken to complete, number of iterations used in each step, and the size of the graph that is used in the positioning step.

5.4 Comparison with Published Results

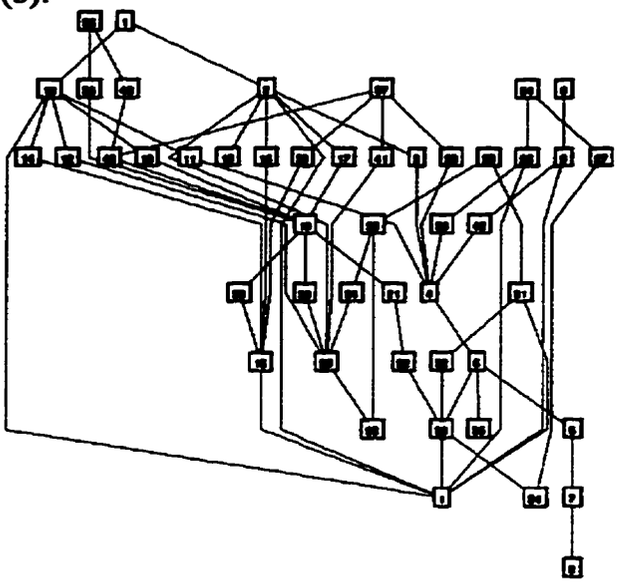
In the literature, most papers on hierarchical directed graph drawing algorithms refer to Sugiyama *et al.*'s paper describing the barycenter and priority methods for drawing acyclic directed graphs. For the purpose of comparing the quality of the drawings of flatdag and dot with Sugiyama's method the reader is referred to Figure 17. Figure 17(a), is taken directly from [STT81], (b) represents the results obtained with using dot, while (c) is the output from flatdag. The duplicated vertex numbers {1,8,24,30,35} are from Sugiyama's original diagram. Even though they are not explained, it appears they represent matching source and sink nodes. The reasons for this labelling is unknown. It should be noted with all layouts in this thesis, that all edges do have direction even though arrows are not on the edges. In all cases, edges are assumed to be pointing from top of the page down, or from left to right in terms of flat edges.

Figure 17

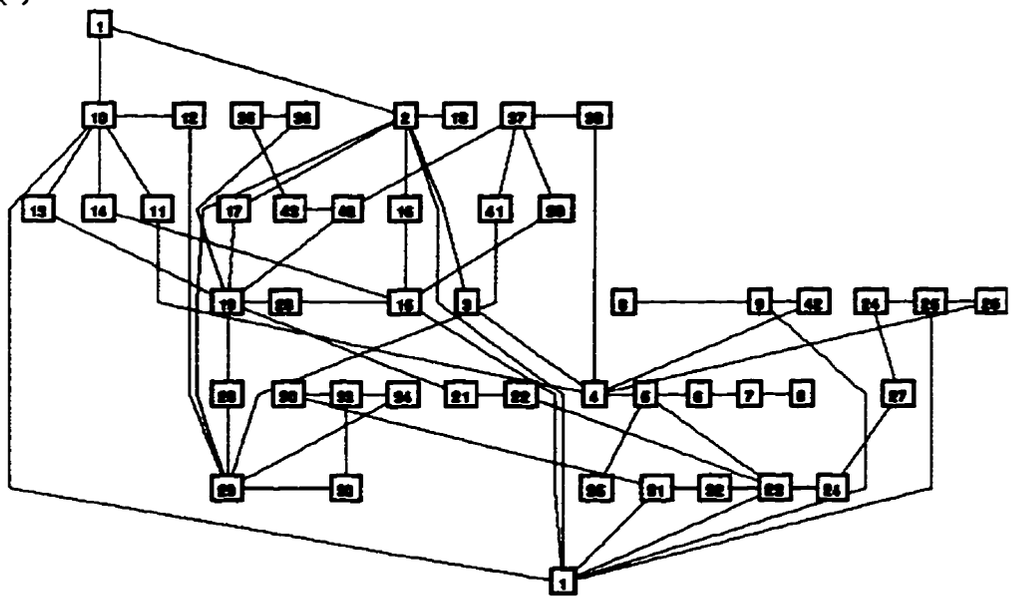
(a).



(b).



(c)



The number of ranks is equal in both (a) and (b), while (c) has two fewer. This also means that the number of virtual nodes is drastically reduced in (c) as compared with (a) and (b). There are 27 virtual nodes compared with 79 and 45 in (a) and (b) respectively. This also leads to the fact that the total sum of the edge length in (c) is much less than in (a) and (b). In (c) the total edge length is 72 while for (a) it is 150 and (b) 112.

Furthermore picture (c) has fewer edge crossings than both (a) and (b), 44 compared with 61 and 67 respectively. Also, try to determine incoming edges to vertex 29. We think this easier to do in Figure 17(c).

The last measured criterion is the relationship between the maximum number of nodes in a rank with respect to the number of ranks. In this case, all three layouts are very similar. (a) has a 2.3:1 ratio while (b) and (c) have a 2.2:1 and 3:1 ratio. On observation however, Figure 1(c) does have a more uniform distribution of vertices per rank compared with (a) and (b).

Therefore according to the measured aesthetic criteria, the layout produced by flatdag is superior to those produced by the other two algorithms because it has less total edge length, fewer edge crossings and a better distribution of vertices than (a) and (b).

Figure 18(a) was also taken from [STT81]. Once again (b) is the resulting layout using dot while (c) is obtained using flatdag. The statistics for these graphs can be seen in Table 1.

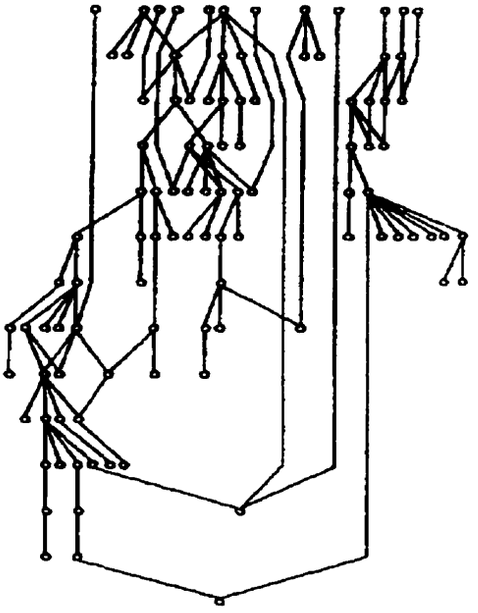
Table 1

Graph	Vertices	Edges	Edge Crossings	Edge Length	Width	Height	Ratio
Fig 18(a)	100	110	16	159	20	14	1.42
Fig 18(b)	100	110	19	120	13	15	0.87
Fig 18(c)	100	110	9	113	27	10	2.7
Fig 18(d)	100	110	16	62	40	6	6.67

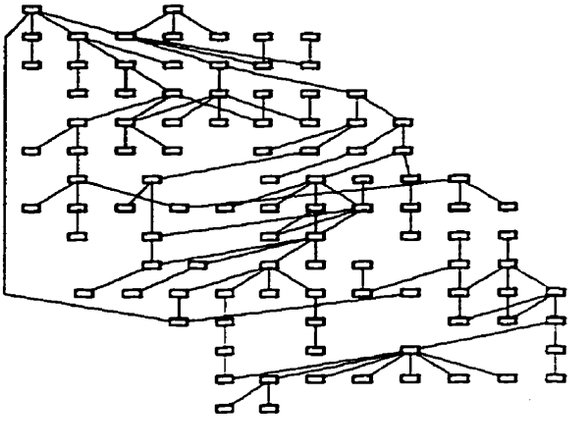
It is easy to see from this table that (b) and (c) have many aesthetic advantages over (a). Especially with respect to the overall edge length of the graph. Notice that the ratio between the number of ranks and the maximum number of vertices in a rank of (c) is 2.7 which is very close to the optimal value of 2.5. The number of edge crossings is also less in (c) than both (a) and (b).

Figure 18

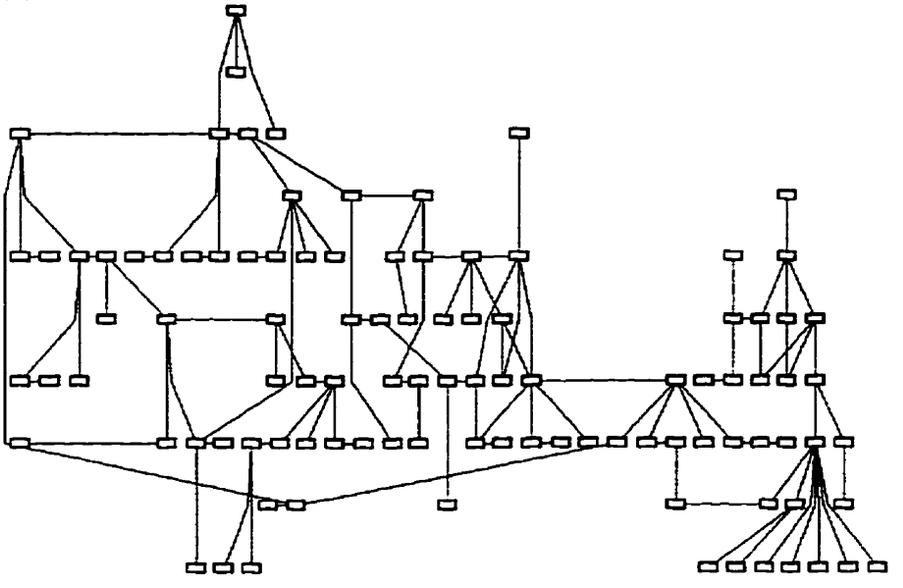
(a)



(b)



(c)



(d)

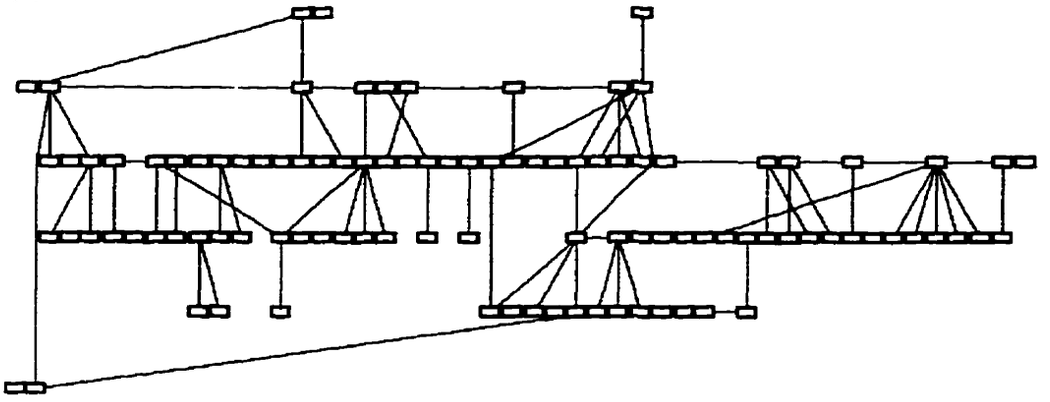


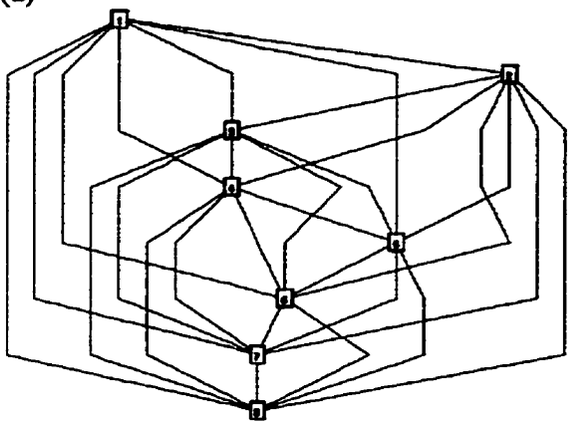
Figure 18(c) however has some imperfections. For example there are two source nodes which have unnecessarily long edges. By placing each of them on the rank directly below, the overall edge length would be reduced by 3. This is caused by the "bumping" heuristic. Recall that the bumping heuristic was designed to compensate for any over-flattening by the flattening heuristic. Figure 18(d) shows the result of flatdag using only the flattening heuristic. Notice that the total edge length has been reduced to only 62 and the number of ranks is 6. This leads to a width to height ratio of over 6. At this point cramping and even overlap among neighboring vertices in the same rank occurs. Thus by using the bumping technique described above to move some vertices down a rank, a better layout can be the result.

In fact this is one of the problems with minimizing the overall edge length. In some instances the number of ranks can be reduced too much. In this example, vertices become crowded and are unable to fit on the page. Figure 17(c) shows where this technique was used in order to augment the assignment of flat edges.

Another significant aspect of Figure 18(d) is the increase in the number of edge crossings. This is an example where by decreasing the number of ranks too far, and thus increasing the number of vertices connected by flat edges, the number of edge crossings will increase. The reasons for this will be described later. For now, however the goal is to minimize the number of ranks without increasing the number of edge crossings significantly.

Figure 19

(a)



(b)

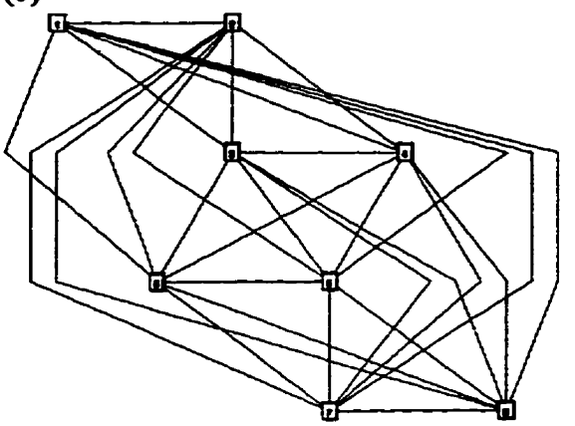
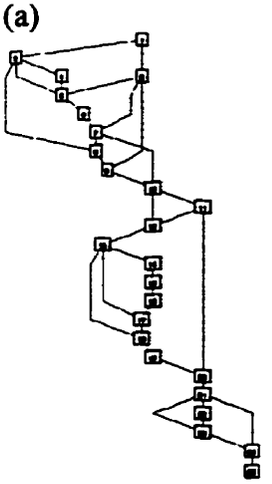


Figure 20

(a)



(b)

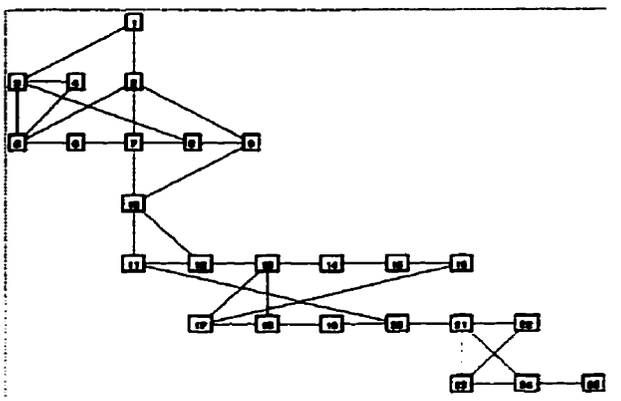
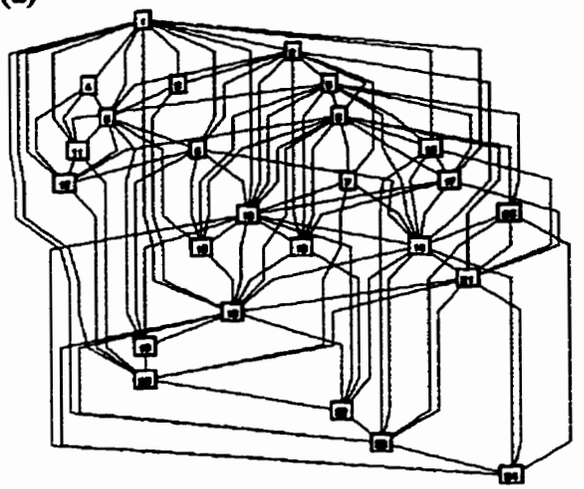


Figure 21

(a)



(b)

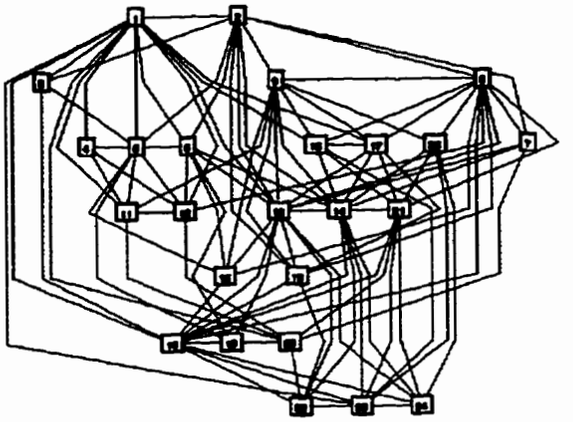
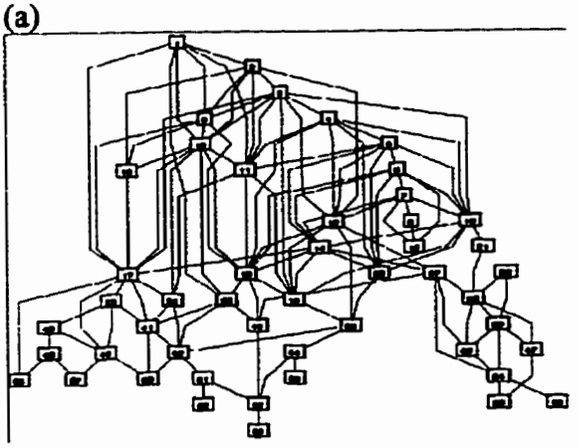


Figure 22

(a)



(b)

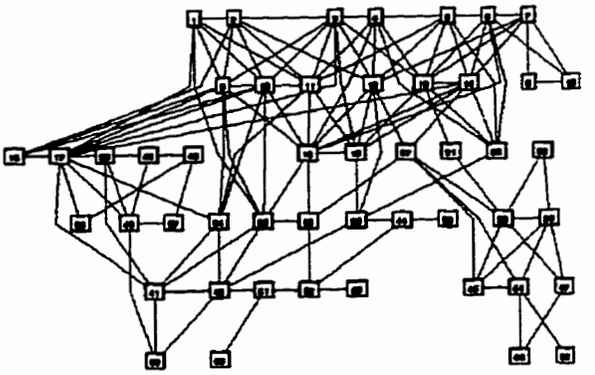
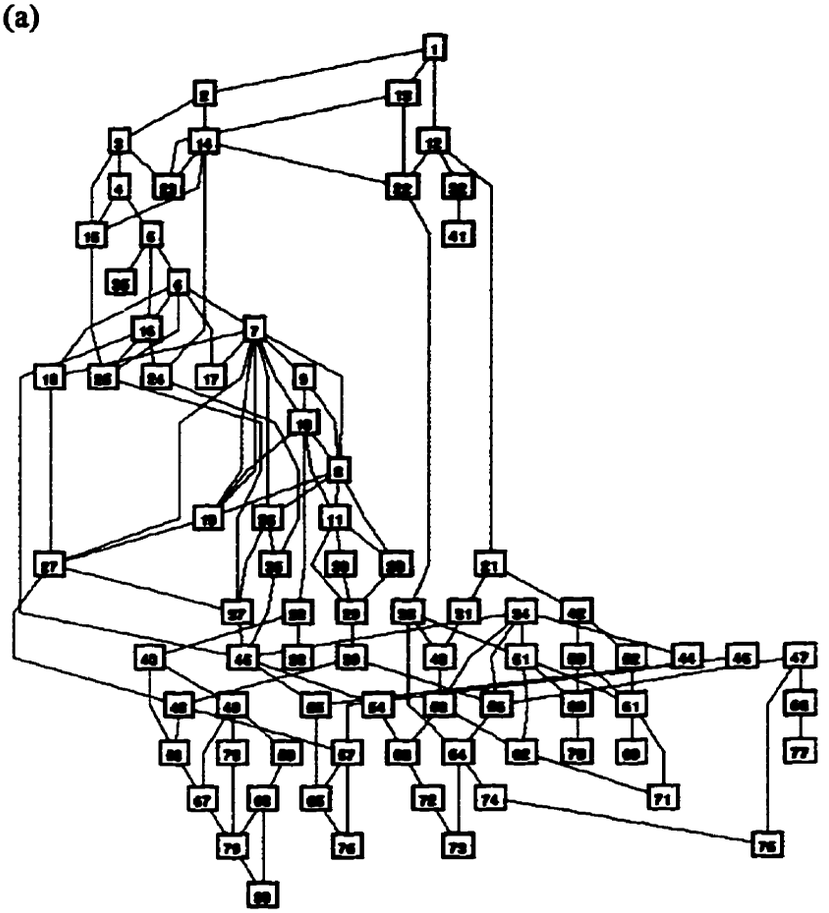


Figure 23

(a)



(b)

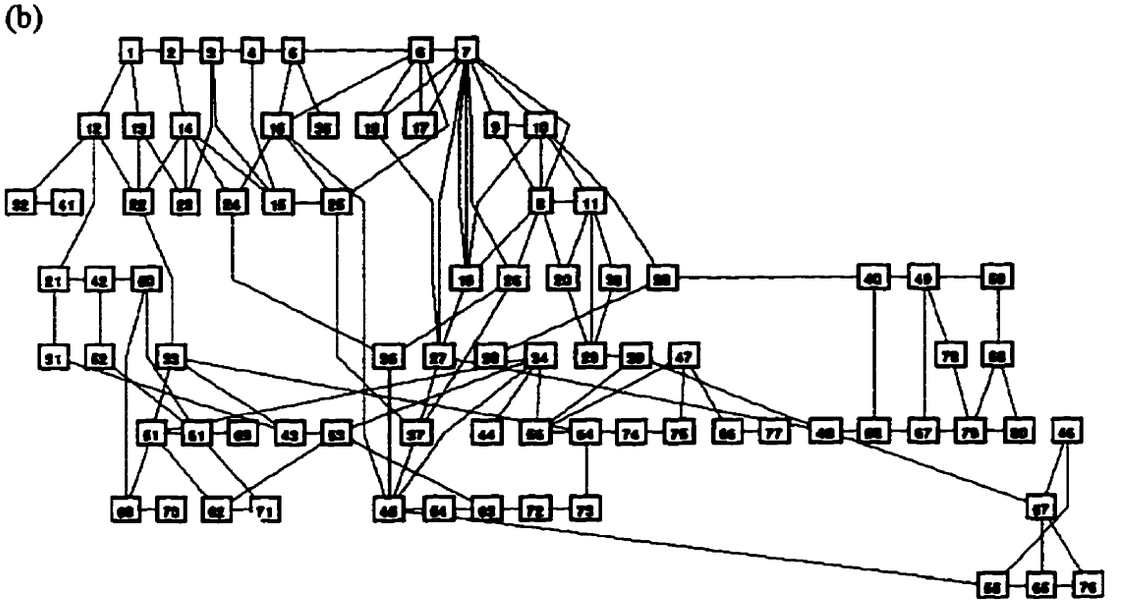
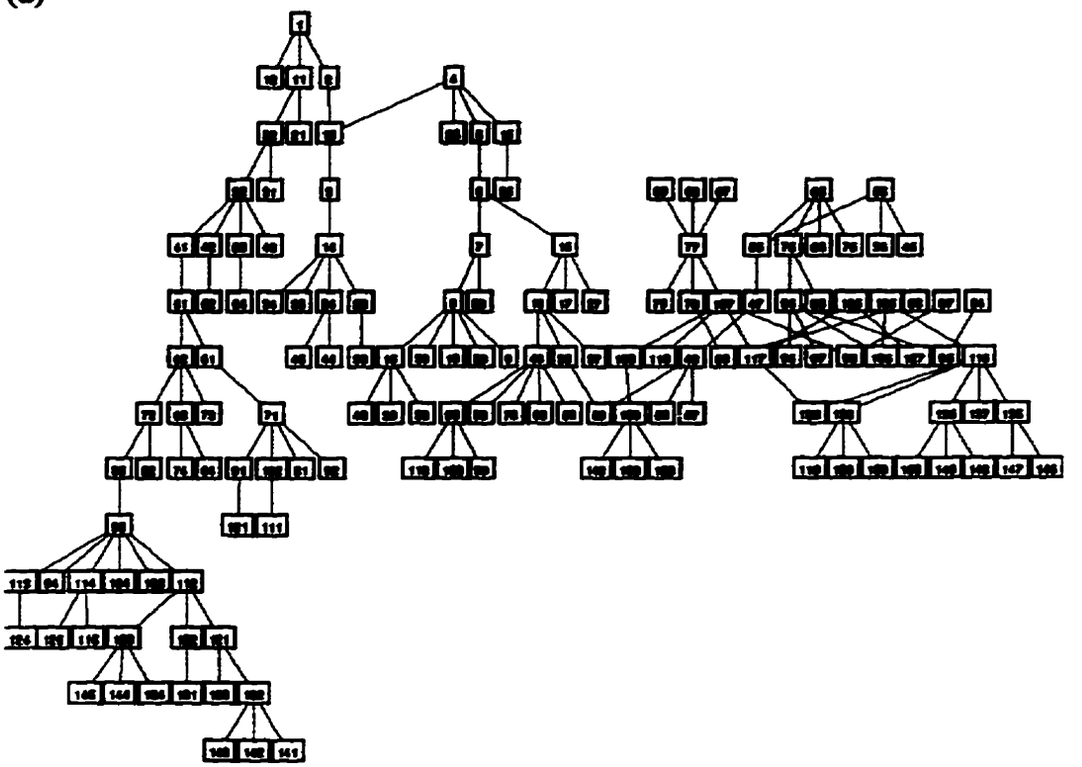
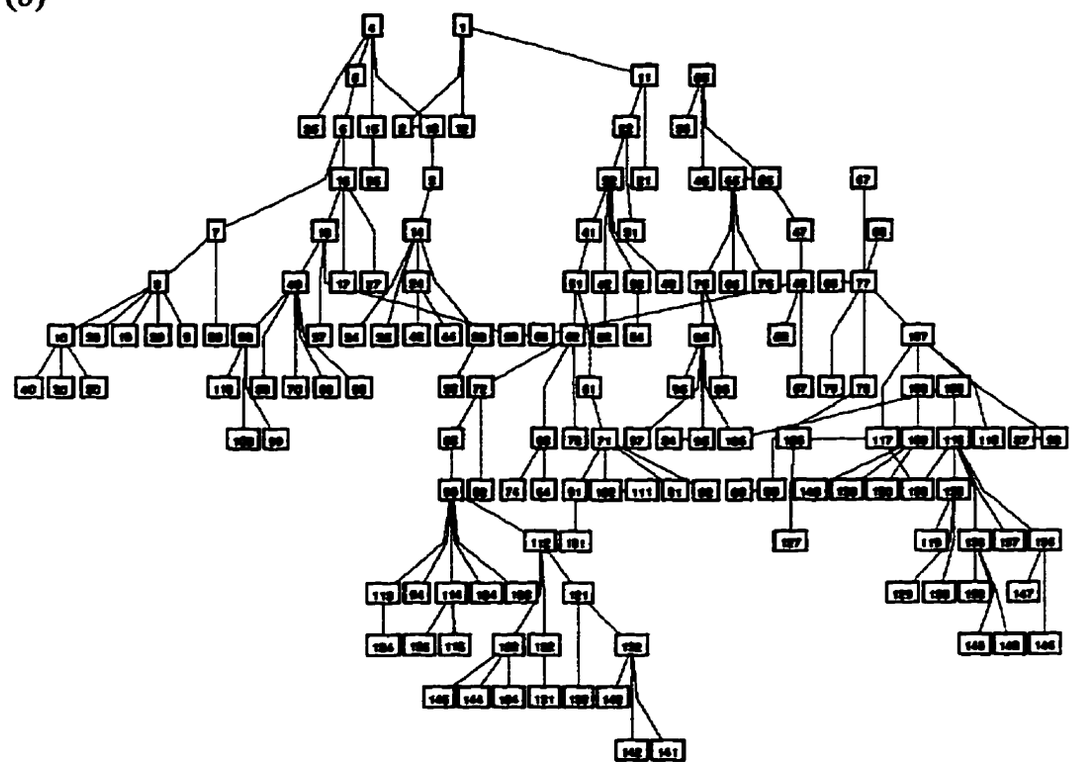


Figure 24

(a)



(b)



5.5 Comparison of Dot and Flatdag

While the previous section looked at results already published by Sugiyama *et al.*, this one looks specifically at the quality of pictures produced by dot and flatdag. A number of different graphs were tested using these two algorithms. The goal was to show the results on a varied set of graph structures. The test data is broken into two sections. The first is a number of hand generated graphs which have different size and densities. The second is a number of randomly generated graphs. These graphs are connected, and have a maximum out-degree of 5. The basic algorithm for creating these graphs was to build a set of N vertices and then do a pass over this set adding one to five out-edges per vertex to some randomly chosen vertices. It was insured that there would be no self-loops or more than one edge connecting any two vertices. This, of course, does not guarantee connected-ness, but a test was done on each graph to ensure that this property held for each randomly generated graph, otherwise it was rejected as a valid test graph.

Because most real life applications that produce directed graphs have some intrinsic underlying structures, the first section attempts to mimic some of these expected results. The second section is used to show how these algorithms might deal with an arbitrary graph. For a listing of the results from the first section the reader may refer to Table 2. Looking at the figures 19-25, the reader will be able to see the final layouts. Here, N is the number of vertices, and M is the number of edges.

Table 2

Part A*

Graph	Algorithm	N	M	Edge Crossings	Edge Length	Width	Height	Ratio
Fig-19a	Dot	8	28	20	84	13	8	1.63
Fig-20a	Dot	25	35	1	69	4	24	0.17
Fig-21a	Dot	25	100	222	358	33	15	2.20
Fig-22a	Dot	60	125	148	310	15	19	0.79
Fig-23a	Dot	80	126	93	223	15	19	0.79
Fig-24a	Dot	150	149	27	150	25	14	1.79
Fig-25a	Dot	180	276	303	452	26	28	0.93

Part B*

Graph	Algorithm	N	M	Edge Crossings	Edge Length	Width	Height	Ratio
Fig-19b	Flatdag	8	28	55	40	10	4	2.50
Fig-20b	Flatdag	25	35	8	19	7	6	1.17
Fig-21b	Flatdag	25	100	336	183	28	7	4.00
Fig-22b	Flatdag	60	125	169	119	18	6	3.00
Fig-23b	Flatdag	80	126	80	133	23	8	2.88
Fig-24b	Flatdag	150	149	18	205	30	15	2.00
Fig-25b	Flatdag	180	276	419	328	36	16	2.25

*This table shows the results of dot on 7 hand created examples.

**This table shows the results of flatdag on the same previous examples.

For a listing of the data regarding the randomly generated graphs, the reader is referred to Table 3 which can be found in the appendix.

5.6 Summary - Layout

From the data there a few noticeable trends. First of all, the number of edge crossings is almost always greater in the layout produced by flatdag than in that by dot. The sum of the total edge length however is almost always less when flatdag is used. The one exception is the graph in Figure 24. It in fact is an anomaly in many cases; for example it has fewer edge crossings and a greater height when processed using flatdag. It is the only example where the number of ranks increase. This also corresponds to an increase in the sum of the overall edge length. Otherwise, the trend is that layouts produced by flatdag, have less height, greater width, more edge crossings, and less overall edge length.

Another observation is that the ratio of width to height is closer to the optimal ratio of 2.5 in virtually every case when flatdag is used. The one exception to this is the graph in Figure 21. Here, the layout is composed predominantly of virtual nodes. The estimate used to determine whether a rank is full is a conservative measure based on the assumption that there is a 2:1 ratio of real vertices compared with virtual ones. Therefore in this situation it is not an accurate enough measure. Also the fact that there are so few vertices in the graph to begin with contributes to the inaccuracy. Notice however that the picture in Figure 21(b) comfortably fits on the page.

The reasons for all these observations will be discussed in chapter 6.

5.7 Efficiency

This section looks at the difference in the overall speed of the two algorithms. Table 4 shows the results of the hand created examples used above while Table 5 shows the results of the algorithms on the 27 randomly generated graphs. For ease of reading, both of these tables appear in the appendix.

The data describes 3 different ways of looking at efficiency. The first is how fast in terms of seconds each stage of the algorithm runs and how fast the algorithm is overall. Secondly, it looks at how the size of the graph changes as it goes through each of the three phases of the algorithms. The third is the number of iterations that the positioning step takes before it comes to completion. All three of these parts express the efficiency of these algorithms from a slightly different angle which is important in determining why one is more efficient than the other.

5.8 Observations - Efficiency

In every case, with the exception of the graph from Figure 24, flatdag was much faster than dot. The ranking steps were approximately equal, but the ordering and position steps were both much faster in flatdag. Also, the size of the auxiliary graph used in the positioning step is much greater when run using dot.

The Rank stage which includes the pre-processing step for flatdag took almost the same amount of time. As the graphs became larger the difference in time became less. The opposite is true for both the ordering and positioning steps. In all cases, with the exception of the graph in Figure 24, ordering and positioning steps were much faster. As the graphs became larger the difference between the speed of dot and flatdag also becomes greater. In fact, in some of the experiments dot takes 10 hours to complete. In comparison, flatdag takes only 6 1/2 minutes.

The reasons for these observations will be answered in the following chapter.

Chapter 6 - Analysis

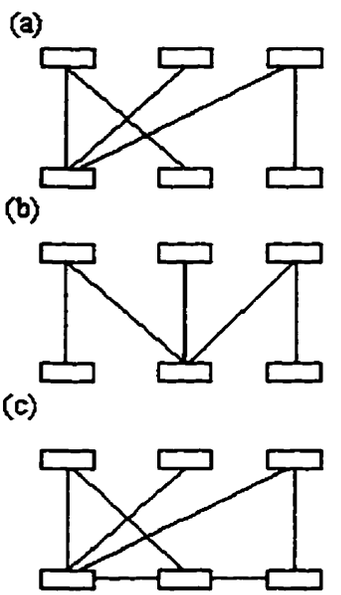
While chapter 5 presented the results, this section will explain some of the observations. First of all, why do some layouts from flatdag have an increase in the number of edge crossings while others have a decrease compared with the layouts produced by dot? Secondly, why is flatdag so much more efficient than dot in most cases?

This section answers these questions as well as many others that pertain to the information gathered in the previous section.

6.1 Difference in Edge Crossings

The cause for the fluctuation in the number of edge crossings is based on the simple structure imposed on the graph by the flat edges. Every flat edge introduces a new constraint to the ranking assignment and the ordering of vertices within each rank. It affects the ranking because one or more levels may be removed from the hierarchy, potentially reducing the number of virtual nodes and edges. Recall, for each virtual node introduced there is at least one more edge introduced into the layout. In fact if an edge of length ϑ is replaced by a connected path of virtual nodes, ϑ edges will inserted into the layout as well as $\vartheta - 1$ virtual nodes. Therefore by reducing the sum of the total edge length, $\Sigma\vartheta$, the probability of having two edges cross is reduced.

Figure 25



A typical example of this can be seen in Figure 17. By placing the source node 35 in (b) into rank 2 from rank 1 (as seen in (c)), two edge crossings are eliminated from the drawing. By reducing the number of ranks in the graph by 1, the length of the two edges decreases. This led to the probability of having fewer edges crossings. Unfortunately this only works to a point before the number of edge crossing begin to increase. This is caused by the inflexible structure of neighboring vertices in a rank that are connected by a flat edge. By constraint, these vertices must be moved around in the ordering as if they are a single node. This ensures that flat edges always point in the same direction, and flat edges do not overlap other vertices. Thus by viewing large chains of vertices connected together by flat edges as single nodes, the number of possible orderings for the vertices in any rank is reduced.

If there are φ vertices in an ordering, then there are $\varphi!$ combinations that they can be placed in order to minimize the number of edge crossings. However, by introducing f flat edges into that ordering, the number of possible combinations becomes $(\varphi-f)!$. This can dramatically reduce the flexibility of the ordering heuristics.

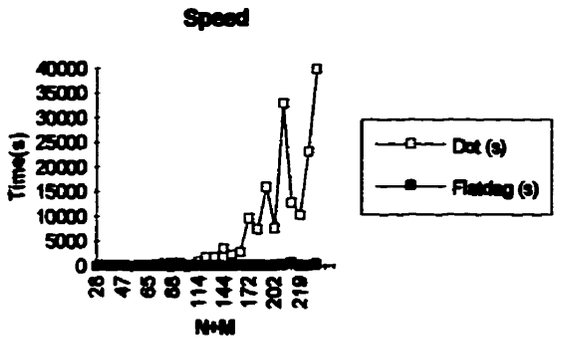
For an example of this, refer to Figure 25. Drawing (a) shows two ranks of 3 vertices each connected by 5 edges. By moving around the vertices in the bottom rank, the number of edge crossings can be reduced to 0. In fact there are 3! or 6 ways of placing vertices in the bottom rank. In contrast Figure 26(c) shows the same configuration as in (a) but with the constraint of having flat edges between the vertices in the bottom row. With the introduction of these flat edges, there is only one possible ordering for the bottom rank. This means an increase of 3 edge crossings for the overall picture.

6.2 Efficiency

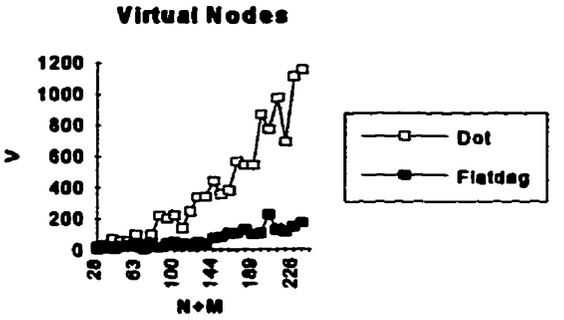
Even though the number of edge crossings increases in many cases, the real difference in the two algorithms is the speed at which they run. The randomly generated graphs in particular showed an enormous difference in terms of efficiency. Graph 1 shows

the change as the size of graphs increase. It is clear that flatdag is dramatically more

Graph 1



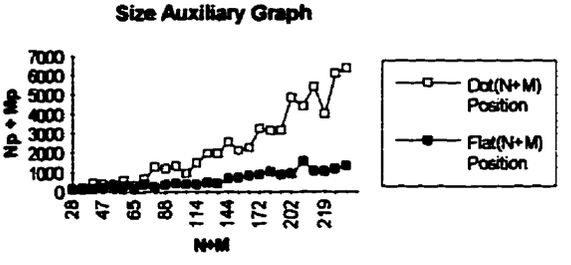
Graph 2



efficient than dot.

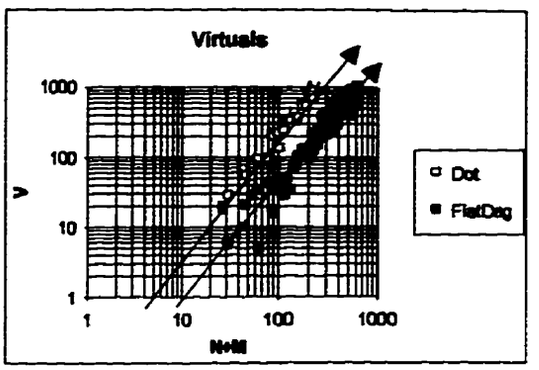
The reason is straightforward. By reducing the number of ranks the sum of the total edge length is also reduced. This in turn reduces the number of virtual nodes. With a graph of size N vertices, M edges, V virtual nodes, and F flat edges, the auxiliary graph used for the positioning step has N + (M-F) + 2V nodes, and N + 2(M - F) + 3V - R edges where R is the number of ranks. Therefore by reducing V, and increasing F the overall size of the auxiliary graph will be decreased. Graph 2 shows the relationship between the number of virtual nodes used during each test, while Graph 3 compares the size of the two auxiliary graphs. Notice how the shape of Graph 2 directly relates to that in Graph 3. Also, the rate at which the size of the auxiliary graph is increasing in dot is much greater than that of flatdag. Furthermore the larger the auxiliary graph, the greater the number of iterations it takes before completion. The combination of these two factors directly will cause both dot and flatdag to slow down on larger graphs. The difference between the two however is that the increase in the size of the auxiliary graph occurs at a much faster rate in dot than in flatdag.

Graph 3



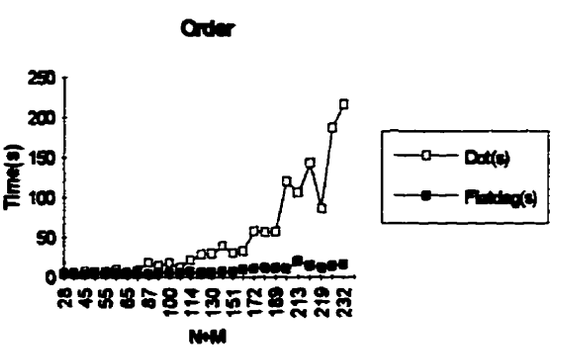
This rate can be measured. By increasing the sample size of the results of flatdag, an interesting observation was made. For the randomly generated graphs, the increase in the difference between the number of virtual nodes used in dot and in flatdag is constant. Evidence of this can be seen in Graph 4. This is a log-log graph of the data in Graph 2, plus some samples of flatdag run on larger graphs. The two straight lines show that the number of virtual nodes used for each algorithm is polynomial. The fact that the lines are parallel indicate that they are both polynomial of the same degree. For these graphs, the number of virtual nodes, V, that are included in the layout can be approximated by the function $V = k\,(N+M)^{1.2}$ where $k = 1$ for flatdag, and $k = 1.45$ for dot. Therefore, the number of virtual nodes used in dot will be approximately 50% greater than in flatdag. In terms of efficiency, this means that the computation time needed to perform the positioning step for a graph of size $N + M$ with dot is equivalent to approximately the same computation time needed for flatdag to compute a layout for a graph of size $2*(N + M)$. Thus adding the preprocess step to dot (as in flatdag) effectively doubles the size of graph that can be processed efficiently.

Graph 4



The key issue here is how large a graph G becomes after it has been transformed into another graph G′ during the ordering and positioning steps. In the ordering step recall that virtual nodes are added to the graph to ensure that all edges have length 1. Depending on the properties of the graph, there can be $O(N^3)$ [FRICK96] of them. Whatever number this is using dot, flatdag reduces this

Graph 5

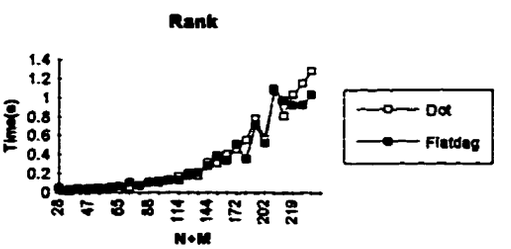


number by a constant factor. In our experiments, this factor is about 50%. Therefore the graph that is transformed in the positioning step, is considerably smaller in the flatdag implementation than in dot. Even though it has not been proven, the efficiency of the positioning step has been observed to be along the lines of an $O(N^4)$ algorithm. Thus, any decrease in the size of N will dramatically increase performance.

It turns out, however, that this is not the only reason for a big performance gain in flatdag over dot. A further speed-up occurs in the ordering step. Graph 5 shows the relationship between dot and flatdag. With fewer vertices to be moved around because of the chains of vertices connected by flat edges, there is less flexibility and fewer changes to be made. The fact that there are 50% fewer virtual nodes also leads to a faster algorithm.

The only advantage in performance that dot has over flatdag occurs occasionally in the ranking step. This happens when vertices are being bumped down ranks. Because it continues until the number of vertices in each rank reaches the required parameters, this step can be a little more time consuming. The ranking stage is by far the fastest part of this algorithm and the increased time spent on bumping is small compared to the rest of the algorithm. However when bumping is unnecessary, flatdag is approximately the same as dot or even faster in some cases. See Graph 6 for the change in efficiency for the ranking step of both implementations.

Graph 6



These arguments on occasion however, can also be reversed. Figure 9 is a classic example where dot works better than flatdag. Because the graph is initially flattened too much, many vertices are bumped, almost all of them. This can lead to more virtual nodes used in flatdag than dot. The efficiency then suffers accordingly. Therefore any graph that requires a large number of bumpings theoretically can result in slower performance for flatdag than dot. This can easily be handled though by doing a simple check. If the number of vertices being bumped is greater than 50% of the total number of vertices in the graph, then assign every edge $\delta(e)$ equal to 1, and restart the entire process without assigning any edges $\delta(e) = 0$.

Then a layout from flatdag will be no different than that of dot in many cases. Of course the efficiency will suffer, but because the pre-processing step takes a very small amount of time compared with the rest of the algorithm, this is acceptable.

6.3 Bumping

Decreasing the number of ranks in any hierarchical layout, is somewhat precarious as seen in Figure 18(c). Thus, bumping is necessary in producing acceptable layouts. However, this is only worthwhile when bumping only a small subset of vertices in the graph. Figure 24 is a perfect example of a graph where too much bumping has occurred. Here, many vertices have been bumped. The result is a layout that is worse than the one produced by dot. It also has more ranks, more virtual nodes and a corresponding increase in the sum of the overall edge length. This process works best when only a small subset of vertices need to be bumped. Graphs that have an ideal hierarchical structure where the width is much greater than height are not necessarily going to be well drawn using flatdag. The flattening part of the preprocess will reduce the number of ranks too much. The bumping part of the preprocess tries to fix this, but is not a good heuristic for wholesale changes. An example where bumping works reasonably well is Figure 18(c). Bumping only a few vertices results in a good layout. In this instance 56 vertices were bumped. As mentioned before however this layout does have imperfections. It turns out that when more than 50% of the vertices are bumped the resulting picture is better when dot is used. This can be easily be incorporated into flatdag by doing a simple check as mentioned earlier. To demonstrate how well the bumping heuristic worked, this check was not incorporated into the implementation of flatdag that produced all the layouts seen in this paper. Therefore, the bumping part of the preprocessing step is sufficient only when small changes need to be made to the results of the flattening step. Otherwise better methods need to be developed to counter act the results of the initial preprocessing step.

Chapter 7 - Conclusions

Flatdag in most cases produces layouts that are equal to, if not better than, those of dot. In all of these cases, the speed of flatdag is far superior to that of dot. Because the speed of these two algorithms is determined predominantly by the positioning step, the number of virtual nodes inserted into the graph must be kept to a minimum without jeopardizing the overall quality of the picture. In instances where the height exceeds the width by a large margin in a layout where no flat edges are used, flatdag produces very good results. However, when the width exceeds the height in the same situation flatdag does not produce results as good as dot. This can be remedied by reverting to dot.

Flatdag produces equal or better layouts at a much faster speed. It has also opened a few doors for more research. The assignment of flat edges in this paper is based on a depth first approach. Preliminary results however show that a breadth-first approach may be better. For instance, by first calculating the longest path for each edge in the graph, then doing a breadth-first traversal starting at the source nodes, there is potential for assigning flat edges in a more intelligent way. Especially if there are more than two edges which have the same value. Perhaps by allowing some sort of look-ahead mechanism, to deduce the immediate surrounding structure of the graph, assignment can be more intelligent.

Another area that can be looked at is the bumping aspect. The heuristic used in this paper, is too simple and produces poor results. It was originally designed to counter-act the possibility that too many edges are assigned a $\delta(e) = 0$. However, it has turned out to be practical in only some instances. Observations of the graphs produced by flatdag without bumping give rise to the possibility that in all cases flatdag, at its worst, could produce the same result as dot, i.e. no edges are initially assigned $\delta(e) = 0$. This, of course, is a continuation of how to decide which edges, if any, should be modified.

This also may lead to help for drawing graphs with excessive width compared to height. In some examples graphs are so wide compared to their height, that dot does not

produce acceptable results either. Adding bumping to dot may also be a way of producing better layouts although the layout will take more time to produce.

This brings us to the positioning part of the algorithm. Because the structure of the transformed graph is much more constrained than in the ranking step, there may be ways of developing short cuts to compute the solution to the linear programming problem. There also may be other approaches to solving this problem that can use the properties of the auxiliary graph to produce more efficient results.

This however, is beyond the scope of this paper and it is presented here as a potential topic for future work. The goal of this paper was to push forward the horizons of the problem of drawing directed graphs. Hopefully, it has accomplished this goal.

Appendix - Tables

Table 3

Part A.

Graph	Algorithm	N	M	Edge Crossings	Edge Length	Width	Height	Ratio
A1(a)	Dot	10	22	12	51	9	8	1.13
A2(a)	Dot	10	18	9	38	7	7	1.00
A3(a)	Dot	15	30	12	99	11	13	0.85
A4(a)	Dot	15	32	19	88	10	12	0.83
A5(a)	Dot	20	35	15	88	14	12	1.17
A6(a)	Dot	20	43	22	138	13	16	0.81
A7(a)	Dot	25	40	12	68	9	9	1.00
A8(a)	Dot	25	53	38	150	18	13	1.38
A9(a)	Dot	30	58	46	256	22	17	1.29
A10(a)	Dot	30	57	62	274	18	22	0.82
A11(a)	Dot	35	65	72	286	19	22	0.86
A12(a)	Dot	35	66	69	205	17	16	1.06
A13(a)	Dot	40	74	66	320	22	23	0.96
A14(a)	Dot	40	79	90	416	24	26	0.92
A15(a)	Dot	45	85	113	422	25	26	0.96
A16(a)	Dot	45	99	255	539	31	25	1.24
A17(a)	Dot	50	101	210	456	31	23	1.35
A18(a)	Dot	50	102	180	482	28	25	1.12
A19(a)	Dot	55	117	293	685	33	29	1.14
A20(a)	Dot	55	118	332	661	37	28	1.32
A21(a)	Dot	65	137	411	1006	43	34	1.26
A22(a)	Dot	65	124	244	666	32	31	1.03
A23(a)	Dot	75	144	412	1121	42	41	1.02
A24(a)	Dot	75	151	478	1266	47	43	1.09
A25(a)	Dot	75	144	357	836	38	37	1.03
A26(a)	Dot	75	138	366	914	44	36	1.22
A27(a)	Dot	75	157	523	1316	46	46	1.00

Part B.

Graph	Algorithm	N	M	Edge Crossings	Edge Length	Width	Height	Ratio
A1(b)	Flatdag	10	22	10	36	10	6	1.67
A2(b)	Flatdag	10	18	9	19	7	4	1.75
A3(b)	Flatdag	15	30	32	31	11	5	2.20
A4(b)	Flatdag	15	32	19	47	9	7	1.29
A5(b)	Flatdag	20	35	22	53	15	7	2.14
A6(b)	Flatdag	20	43	38	67	13	7	1.86
A7(b)	Flatdag	25	40	29	34	10	6	1.67
A8(b)	Flatdag	25	53	59	83	18	7	2.57
A9(b)	Flatdag	30	58	106	81	21	7	3.00
A10(b)	Flatdag	30	57	78	55	15	5	3.00
A11(b)	Flatdag	35	65	78	96	19	8	2.38
A12(b)	Flatdag	35	66	105	87	22	6	3.67
A13(b)	Flatdag	40	74	125	77	22	6	3.67
A14(b)	Flatdag	40	79	131	105	24	7	3.43
A15(b)	Flatdag	45	85	170	92	23	6	3.83
A16(b)	Flatdag	45	99	320	147	29	7	4.14
A17(b)	Flatdag	50	101	287	159	30	9	3.33
A18(b)	Flatdag	50	102	212	180	27	10	2.70
A19(b)	Flatdag	55	117	397	191	34	10	3.40
A20(b)	Flatdag	55	118	452	220	39	10	3.90
A21(b)	Flatdag	65	137	513	203	40	9	4.44
A22(b)	Flatdag	65	124	333	186	34	10	3.40
A23(b)	Flatdag	75	144	483	226	42	11	3.82
A24(b)	Flatdag	75	151	588	256	46	10	4.60
A25(b)	Flatdag	75	144	516	216	38	10	3.80
A26(b)	Flatdag	75	138	451	319	44	13	3.38
A27(b)	Flatdag	75	157	653	286	42	11	3.82

Table 4

Part A

Graph	Algorithm	N	M	V*	Rank(s)	Order(s)	Position(s)	Time(s)	N(Position)	M(Position)	I
Fig-19a	Dot	8	28	56	0.01	5.2	9.59	14.87	148	224	53
Fig-20a	Dot	25	35	33	0.05	12.4	4.07	16.65	127	172	31
Fig-21a	Dot	25	100	257	0.14	17.44	797.26	815.19	640	983	280
Fig-22a	Dot	60	125	186	0.32	26.44	474.05	501.07	545	839	219
Fig-23a	Dot	80	126	92	0.73	16.77	136.87	154.66	395	599	134
Fig-24a	Dot	150	149	0	2.35	13.97	36.23	55.05	300	436	57
Fig-25a	Dot	180	276	173	8.22	33.41	1149.63	1192.76	805	1229	254

Part B

Graph	Algorithm	N	M	V*	Rank(s)	Order(s)	Position(s)	Time(s)	N(Position)	M(Position)	I
Fig-19b	Flatdag	8	28	16	0.03	3.32	1.24	4.65	64	100	14
Fig-20b	Flatdag	25	35	0	0.06	3.03	0.47	3.64	44	56	2
Fig-21b	Flatdag	25	100	94	0.14	6.84	63.19	70.35	302	478	94
Fig-22b	Flatdag	60	125	20	0.34	11.52	11.31	23.38	188	301	47
Fig-23b	Flatdag	80	126	37	0.89	9.26	25.56	35.96	250	375	69
Fig-24b	Flatdag	150	149	210	2.65	16.47	173.49	193.13	420	610	147
Fig-25b	Flatdag	180	276	115	11.23	24.53	524.23	561.45	623	935	188

* V represents the number of virtual nodes inserted into the layout of the graph

* I represents the number of iterations used in the position step.

Table 5

Part A

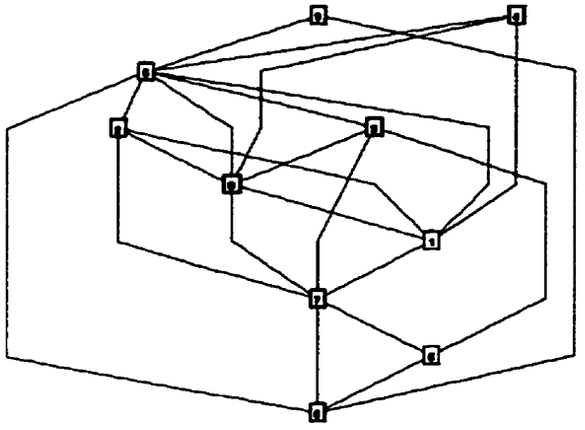
Graph	Algorithm	V	Rank(s)	Order(s)	Position(s)	Time(s)	N(Position)	M(Position)	I
A1(a)	Dot	29	0.01	5.13	2.56	7.8	90	133	31
A2(a)	Dot	20	0.02	4.53	1.34	5.95	68	99	22
A3(a)	Dot	69	0.02	7.54	17.84	25.5	183	269	72
A4(a)	Dot	56	0.02	6.93	9.7	16.8	159	235	48
A5(a)	Dot	53	0.02	7	9.2	16.33	161	237	44
A6(a)	Dot	95	0.04	9.58	47.04	56.83	253	375	101
A7(a)	Dot	29	0.03	5.83	5.39	11.35	122	181	40
A8(a)	Dot	97	0.04	9.1	69.2	78.5	272	409	131
A9(a)	Dot	198	0.1	14.58	394.06	409	484	723	244
A10(a)	Dot	217	0.08	17.87	514.8	533	521	773	268
A11(a)	Dot	221	0.1	17.96	513.5	531.92	542	806	243
A12(a)	Dot	139	0.13	11.91	157.84	170.06	379	568	154
A13(a)	Dot	246	0.17	21.66	795.6	818.06	606	903	300
A14(a)	Dot	337	0.2	28.81	1646	1676	793	1183	388
A15(a)	Dot	337	0.18	29.96	1691	1722.6	804	1200	396
A16(a)	Dot	440	0.32	39.21	3600	3640	1024	1538	506
A17(a)	Dot	365	0.31	30.46	2200	2231	861	1294	445
A18(a)	Dot	380	0.4	32.8	2790	2825	912	1369	450
A19(a)	Dot	568	0.46	58.29	9540	9599	1308	1964	702
A20(a)	Dot	543	0.55	56.57	7347	7420	1259	1892	591
A21(a)	Dot	869	0.56	120.5	15790	15992	1940	2912	1152
A22(a)	Dot	542	0.78	57.79	7580	7640	1273	1908	742
A23(a)	Dot	977	0.81	142.67	23004	23150	2173	3252	1196
A24(a)	Dot	1115	1.15	186.9	32700	32900	2456	3679	1376
A25(a)	Dot	692	1.04	86.17	10260	10348	1603	2402	874
A26(a)	Dot	776	1.08	105.74	12600	12750	1765	2643	1022
A27(a)	Dot	1159	1.29	215.8	39600	39820	2550	3820	1464

Part B

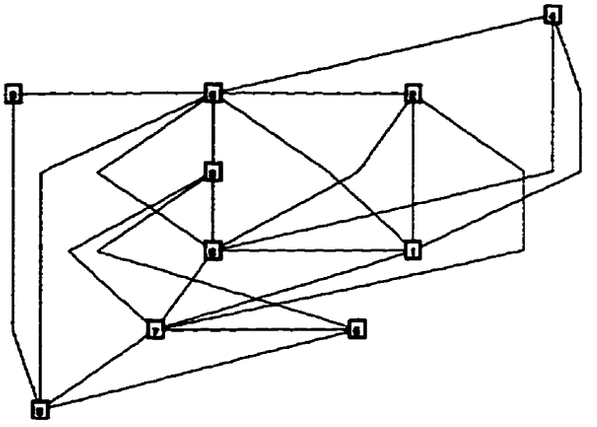
Graph	Algorithm	V	Rank(s)	Order(s)	Position(s)	Time(s)	N(Position)	M(Position)	I
A1(b)	Flatdag	18	0.03	4.08	1.13	5.31	64	94	12
A2(b)	Flatdag	6	0.05	3.47	0.48	4.09	35	50	3
A3(b)	Flatdag	10	0.04	3.84	0.9	4.85	56	82	11
A4(b)	Flatdag	21	0.04	4.72	1.73	6.6	83	123	16
A5(b)	Flatdag	28	0.05	4.77	2.88	7.78	101	147	28
A6(b)	Flatdag	35	0.05	5.68	5.25	11.07	122	182	38
A7(b)	Flatdag	5	0.06	4.4	1.09	5.65	64	92	14
A8(b)	Flatdag	43	0.1	5.65	8.41	14.3	151	227	44
A9(b)	Flatdag	39	0.11	5.58	7.31	13.12	150	224	38
A10(b)	Flatdag	16	0.07	4.21	2.5	6.89	101	151	20
A11(b)	Flatdag	49	0.12	6.15	11.17	17.58	180	268	43
A12(b)	Flatdag	37	0.12	4.93	7.76	12.96	159	240	33
A13(b)	Flatdag	29	0.13	6.83	6.67	13.8	146	217	34
A14(b)	Flatdag	48	0.18	6.1	14.67	21.09	193	291	49
A15(b)	Flatdag	35	0.21	6.19	10.25	16.81	172	258	40
A16(b)	Flatdag	74	0.28	7.61	43.44	51.52	266	406	82
A17(b)	Flatdag	83	0.39	7.49	58.36	67.45	292	442	94
A18(b)	Flatdag	105	0.34	9.65	83.93	94.14	335	505	103
A19(b)	Flatdag	107	0.51	10.83	112.57	124.15	353	534	122
A20(b)	Flatdag	131	0.35	11.21	99.9	111.72	406	616	83
A21(b)	Flatdag	104	0.52	10.34	124.18	135.35	372	566	121
A22(b)	Flatdag	98	0.72	11.25	75.1	87.38	349	525	84
A23(b)	Flatdag	129	0.97	14.2	146.67	162.18	430	645	115
A24(b)	Flatdag	147	0.92	13.53	279.18	294.02	478	724	171
A25(b)	Flatdag	113	0.92	11.51	113.04	125.82	404	610	95
A26(b)	Flatdag	225	1.1	19.78	702.4	723.73	619	925	257
A27(b)	Flatdag	174	1.04	15.62	369.16	386.18	535	810	181

Appendix - Layouts of Random Graphs

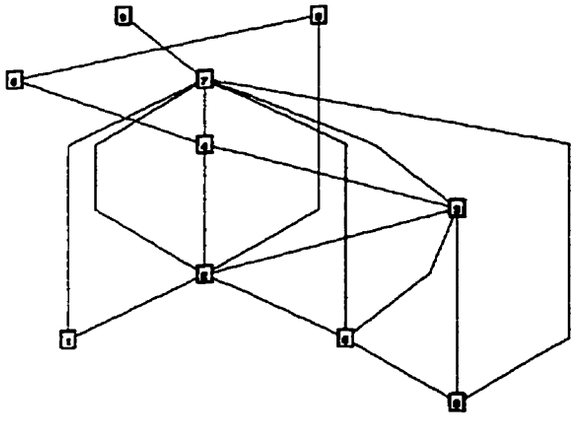
A 1 (a).



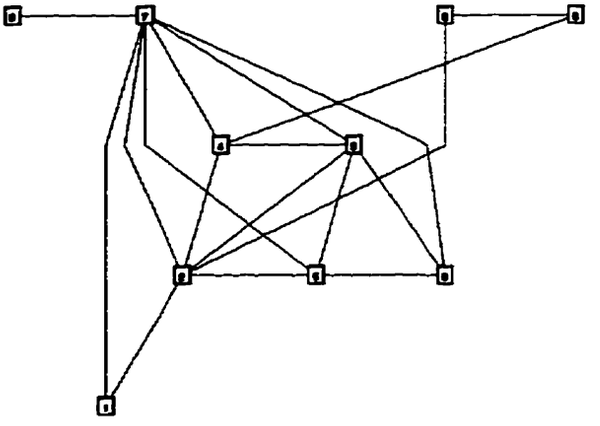
(b)



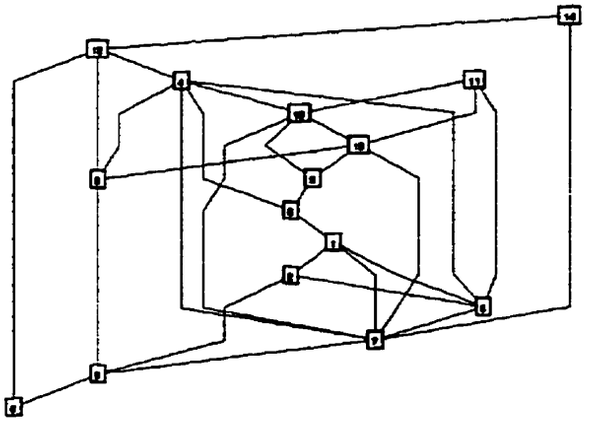
A 2 (a)



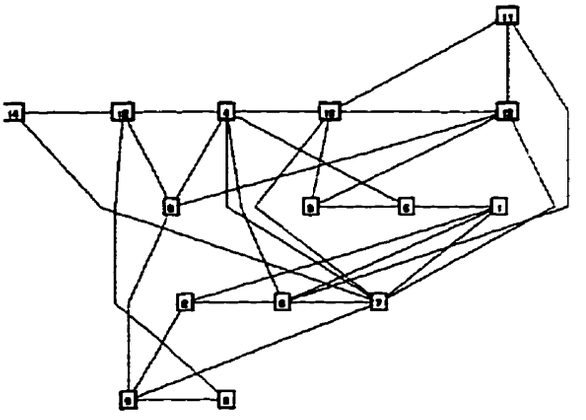
(b)



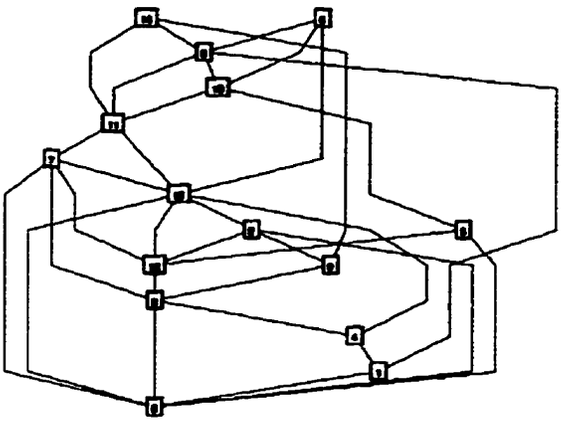
A 3 (a)



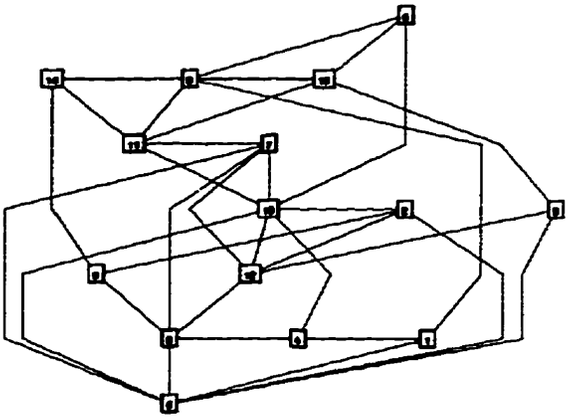
(b)



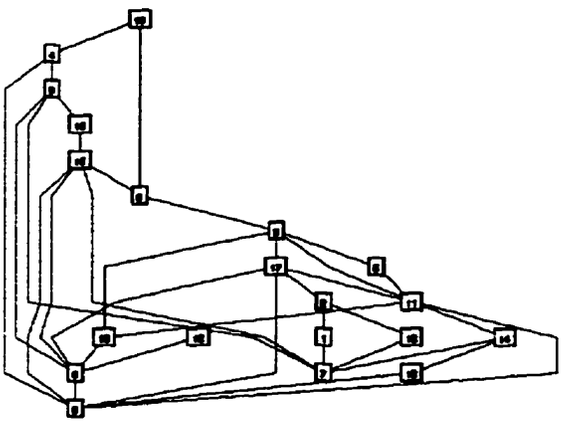
A 4 (a)



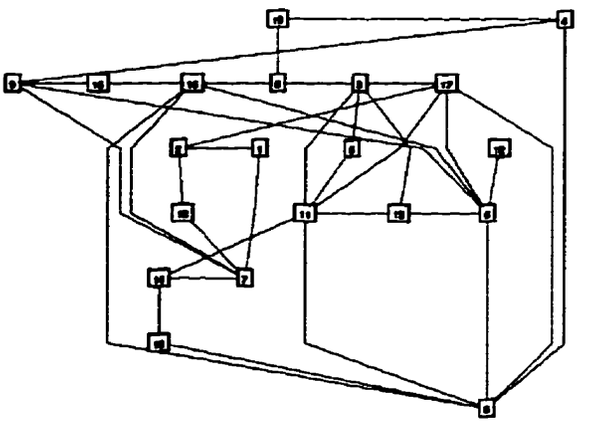
(b)



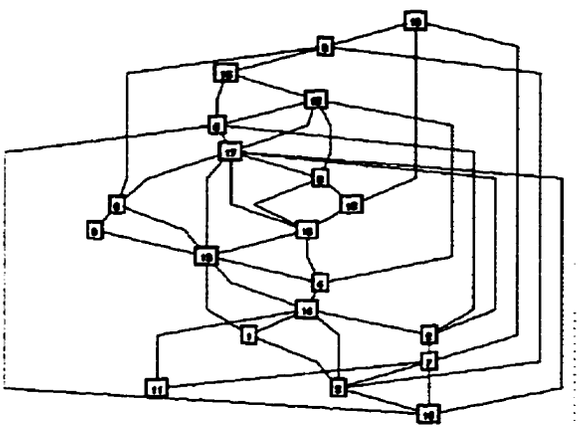
A 5 (a)



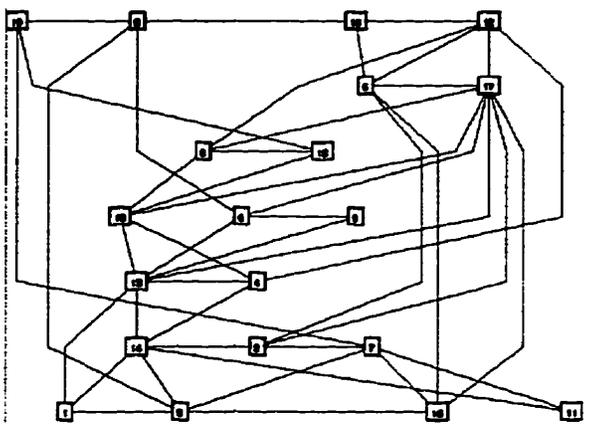
(b)



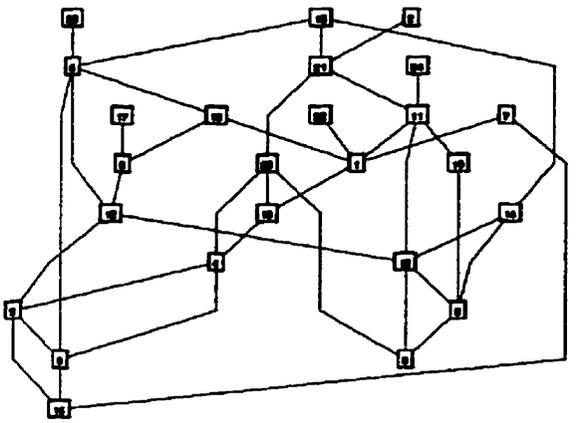
A 6 (b)



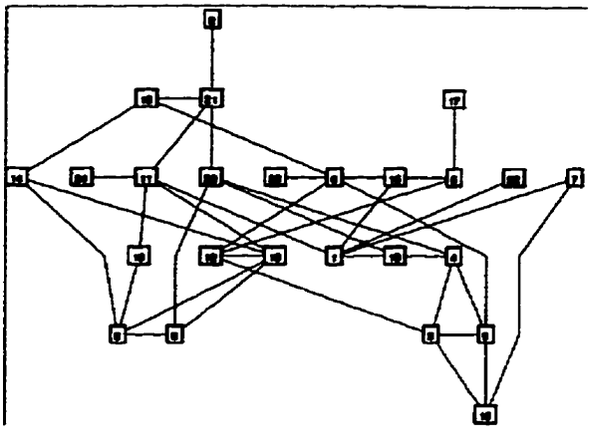
(b)



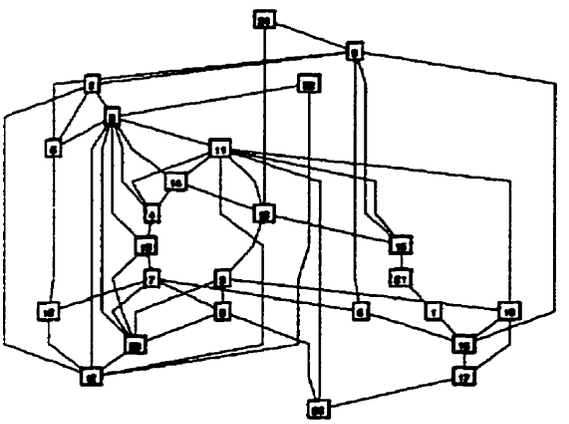
A 7 (a)



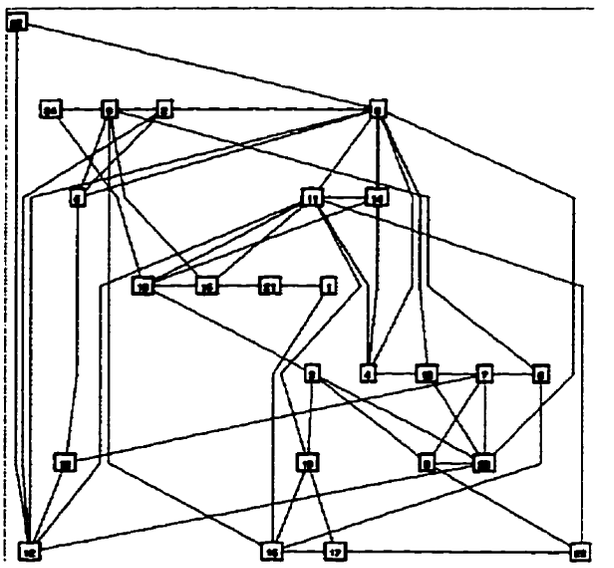
(b)



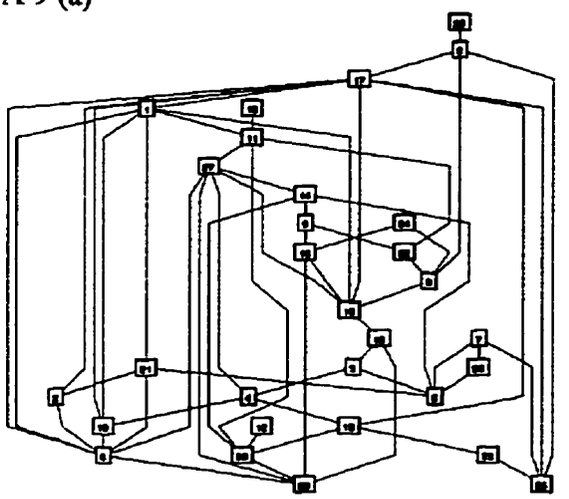
A 8 (a)



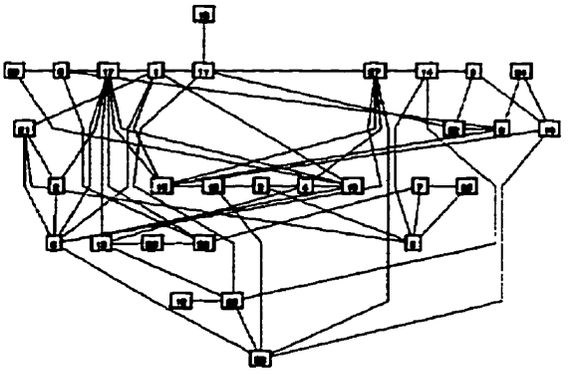
(b)



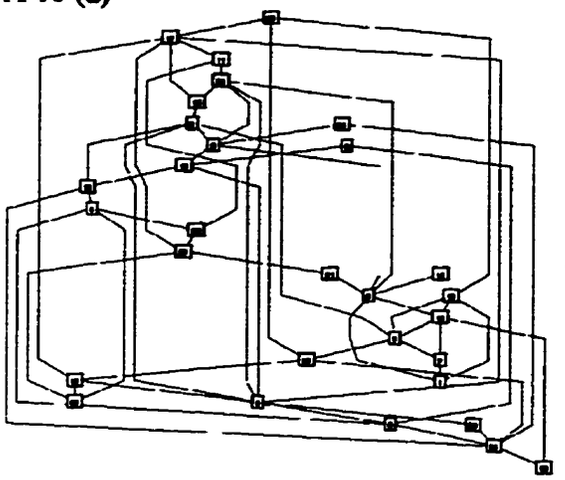
A 9 (a)



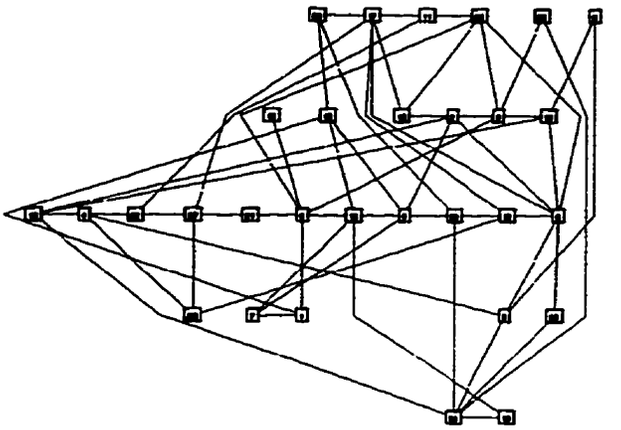
(b)



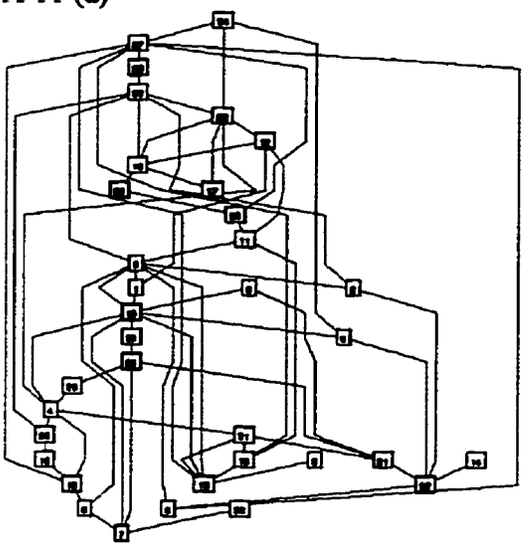
A 10 (a)



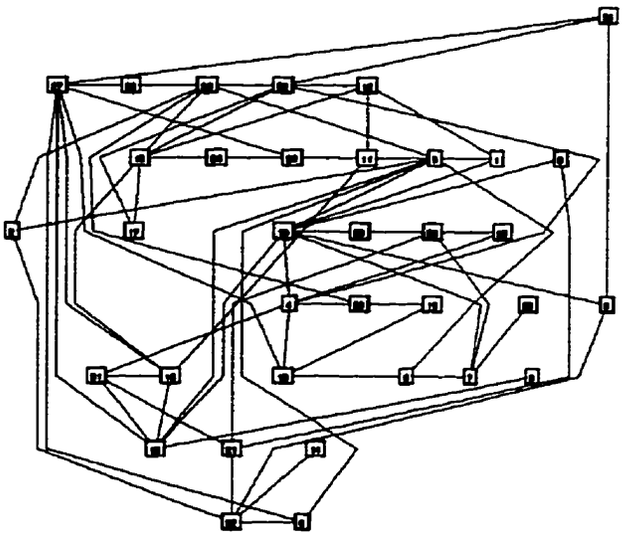
(b)



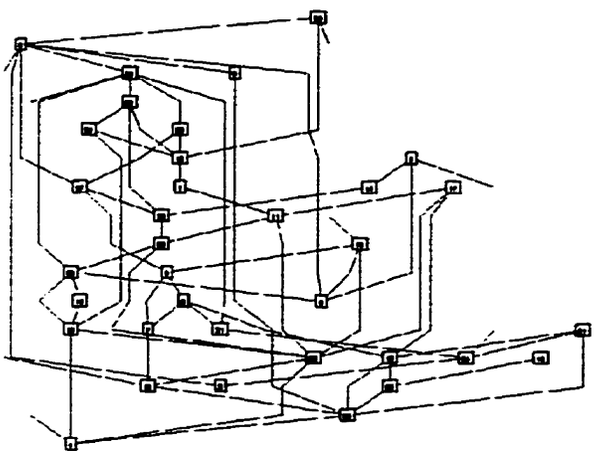
A 11 (a)



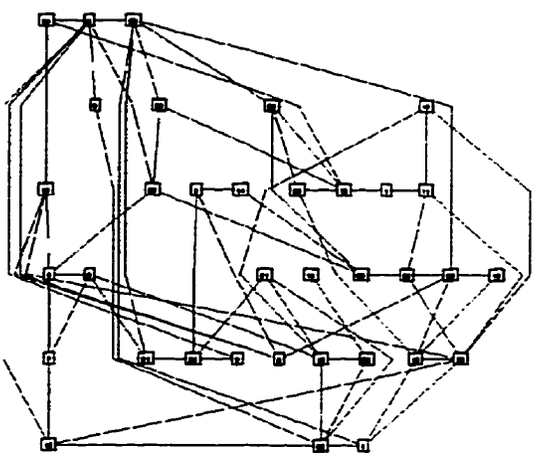
(b)



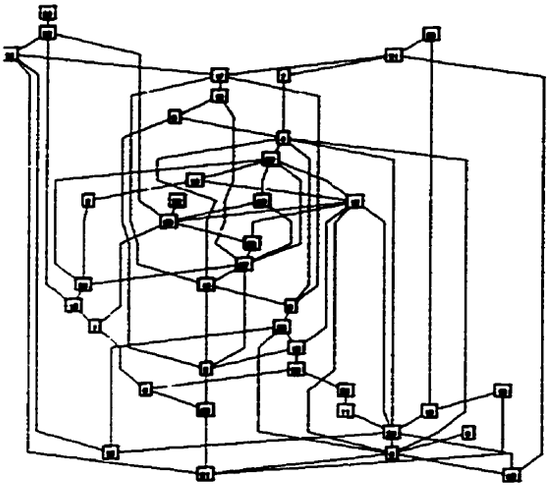
A 12 (a)



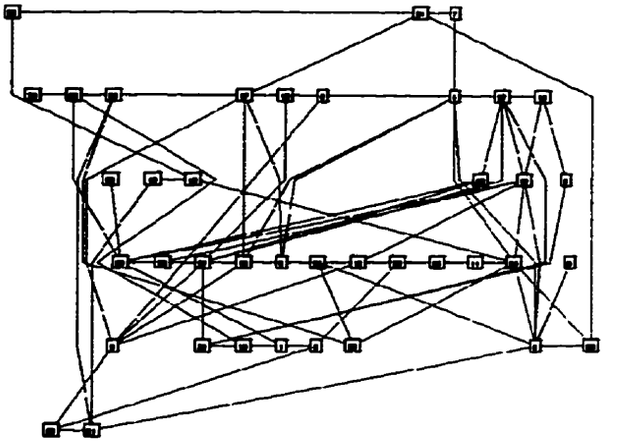
(b)



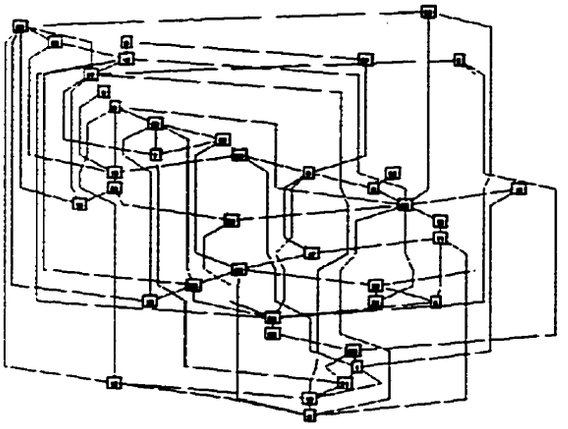
A 13 (a)



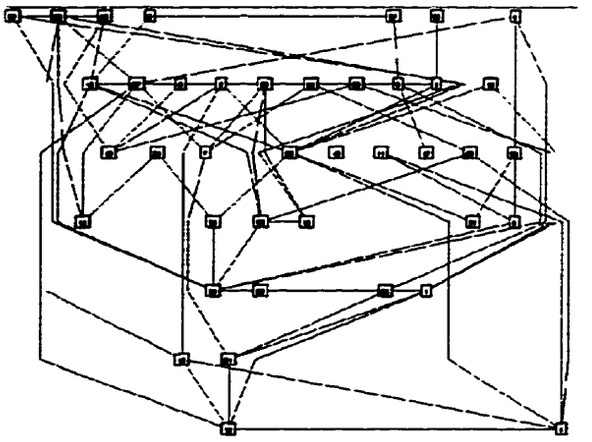
(b)



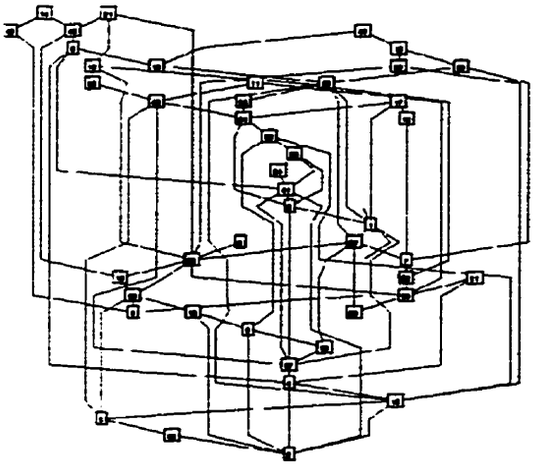
A 14 (a)



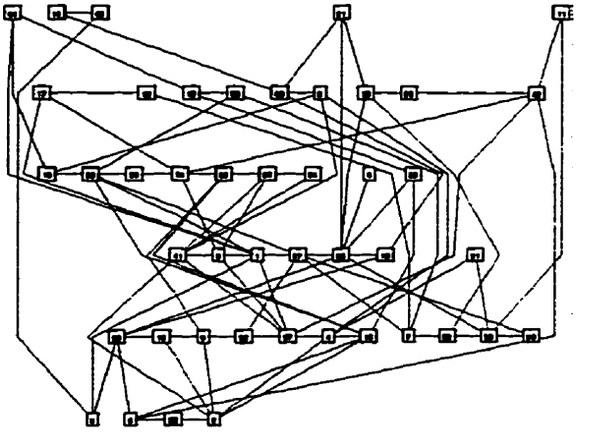
(b)



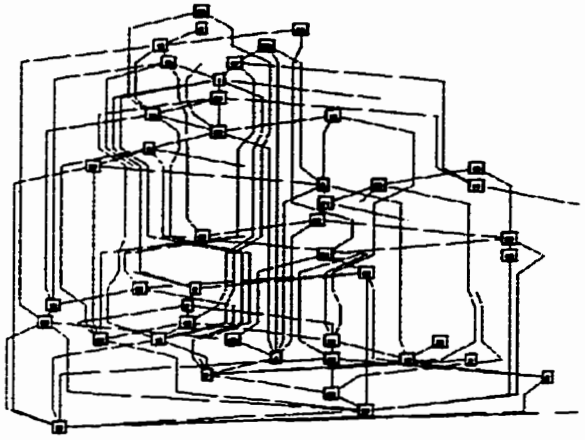
A 15 (a)



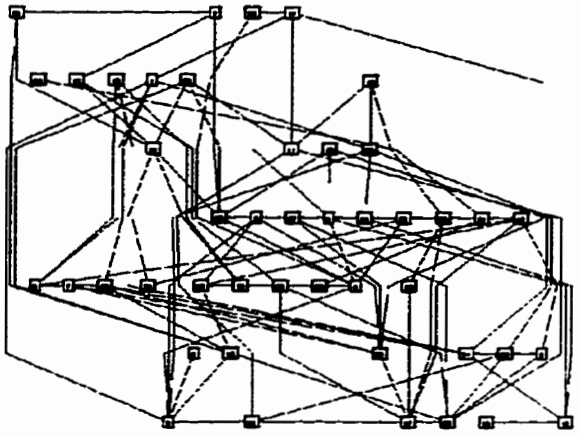
(b)



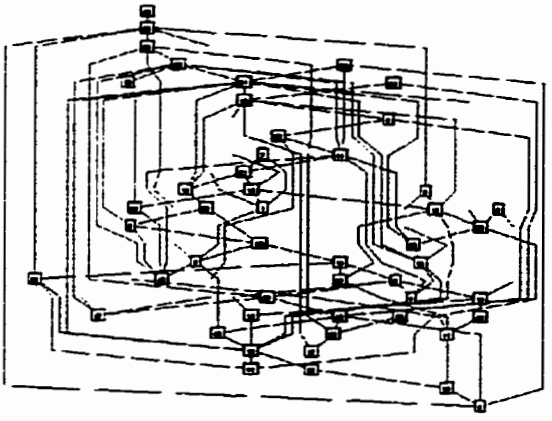
A 16 (a)



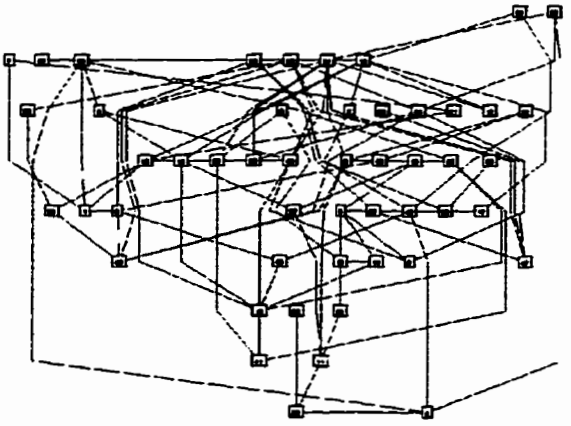
(b)



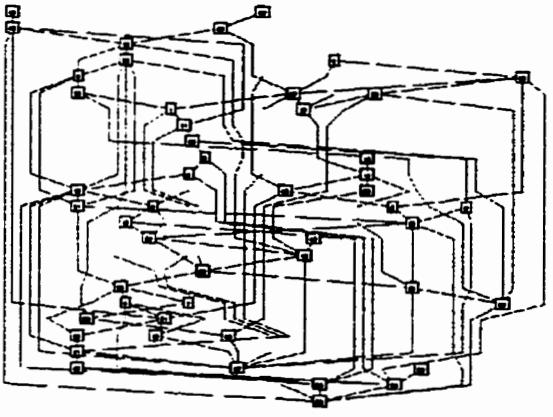
A 17 (a)



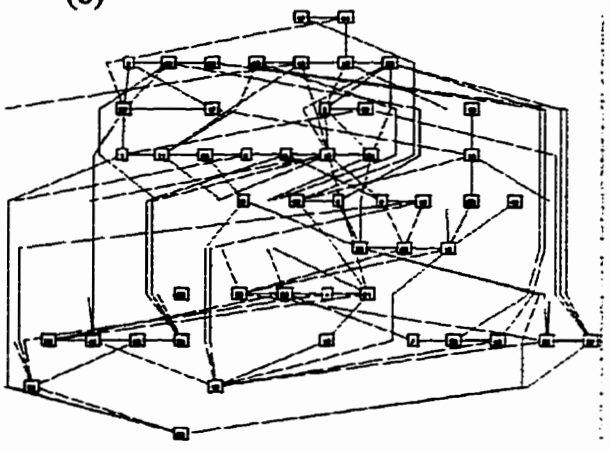
(b)



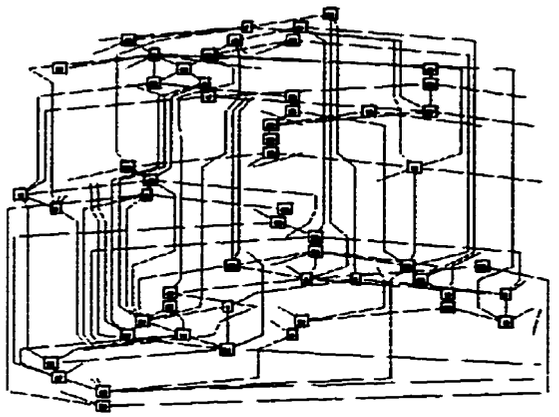
A 18 (a)



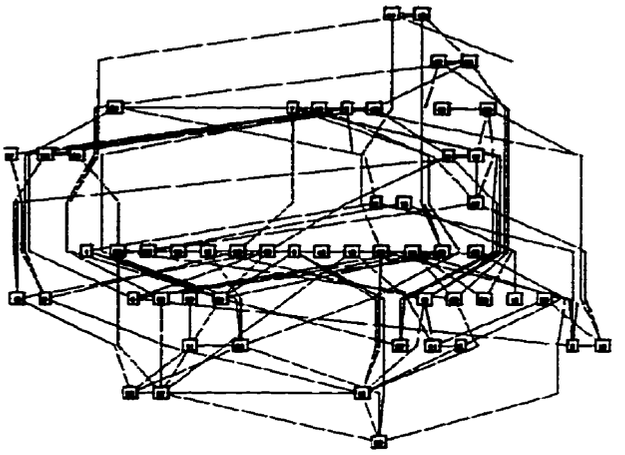
(b)



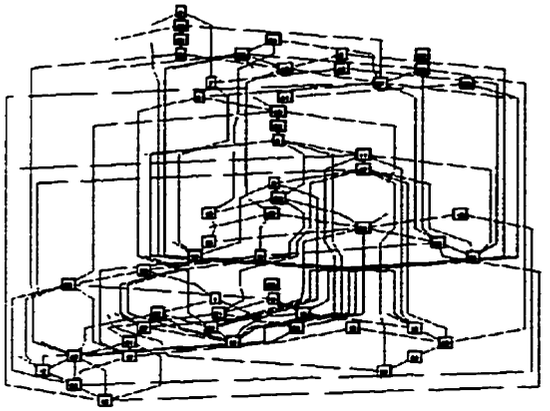
A 19 (a)



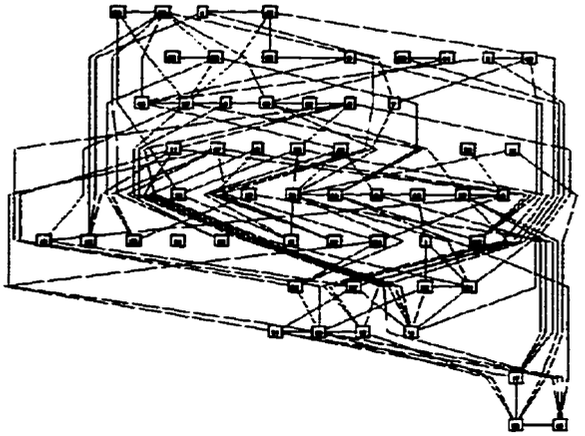
(b)



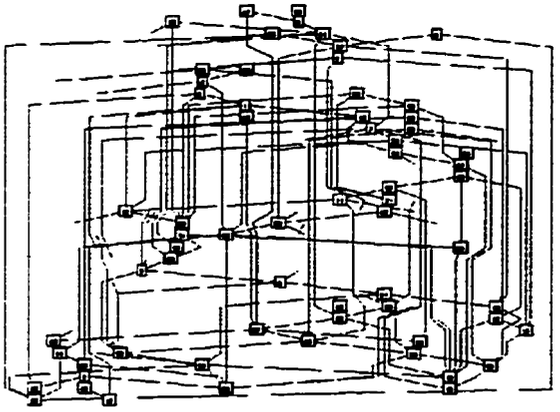
A 20 (a)



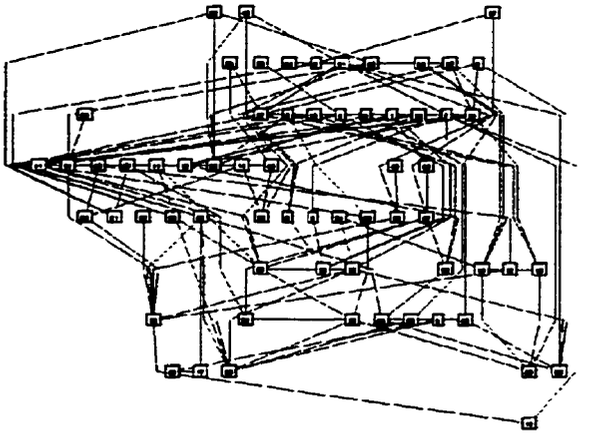
(b)



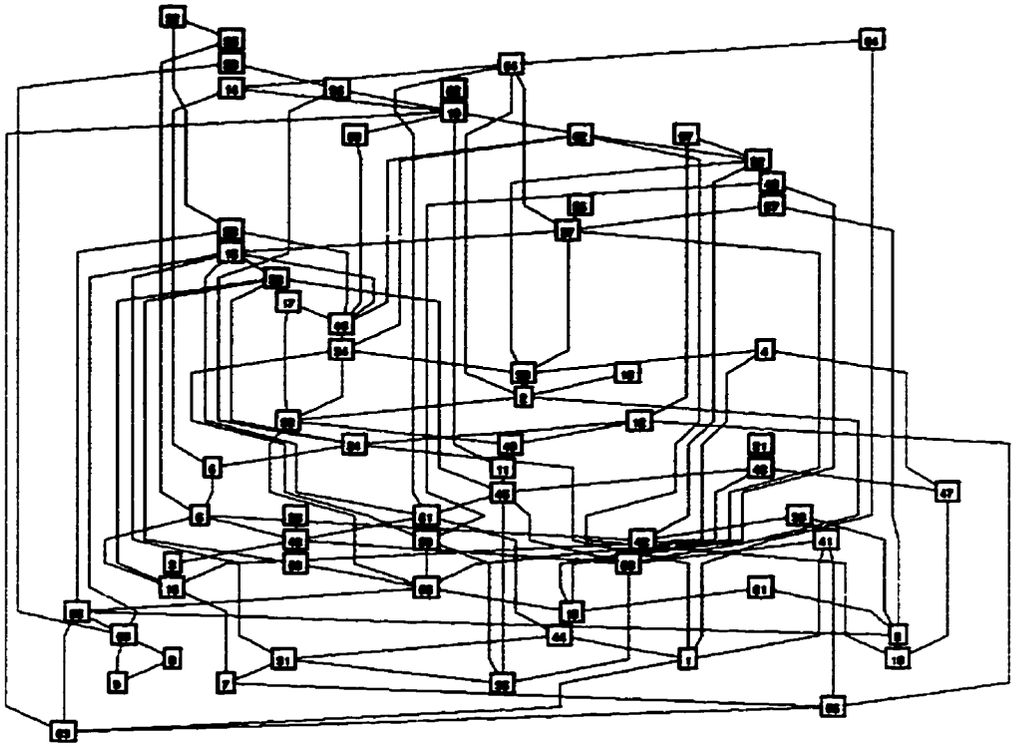
A 21 (a)



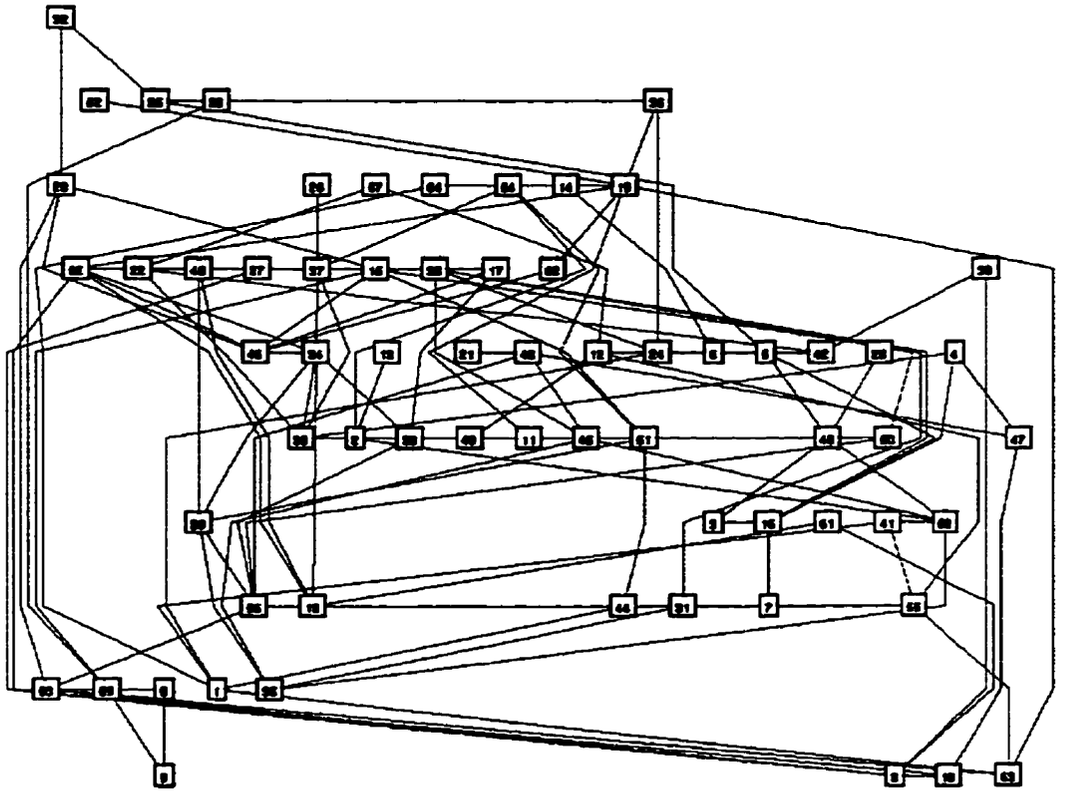
(b)



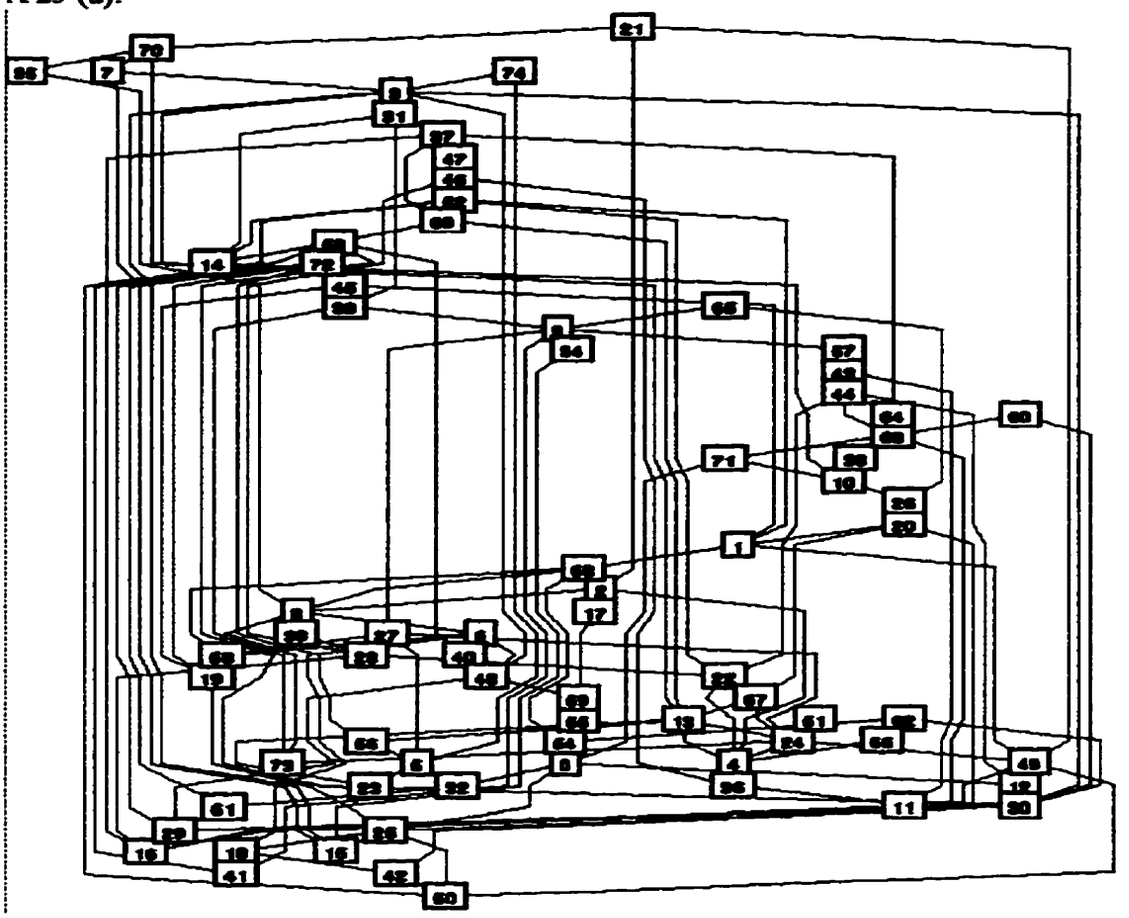
A 22 (a)



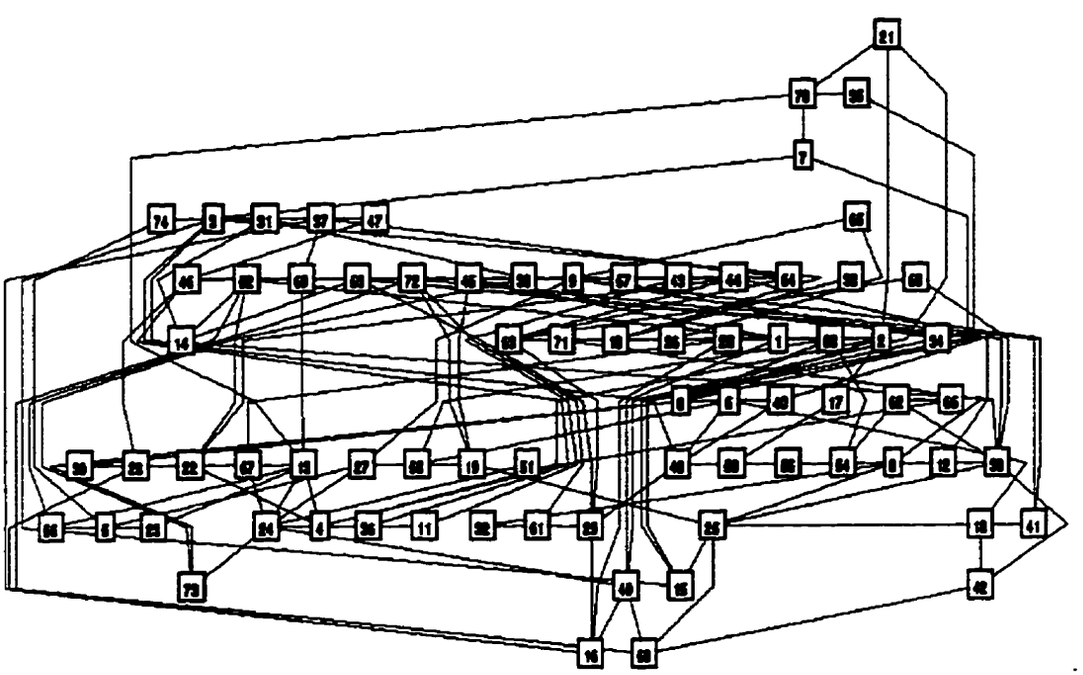
A 22 (b)



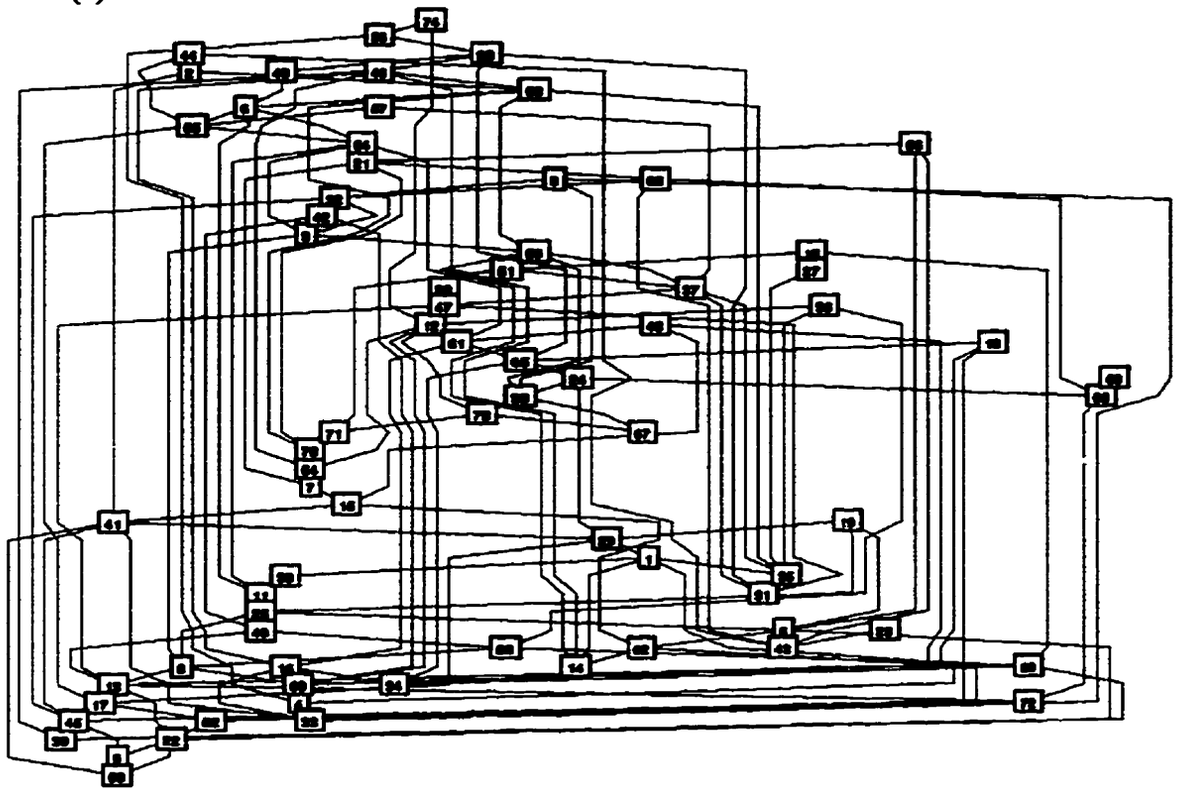
A 23 (a).



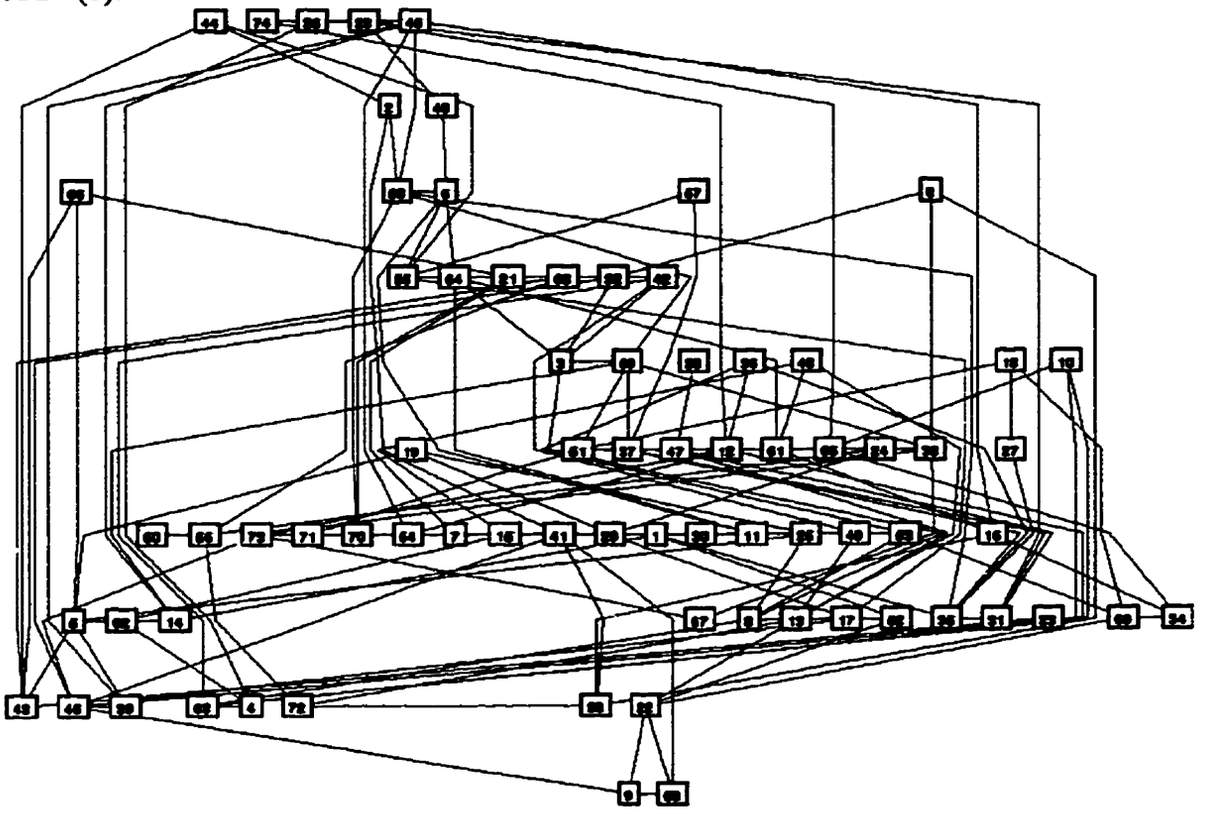
A 23 (b).



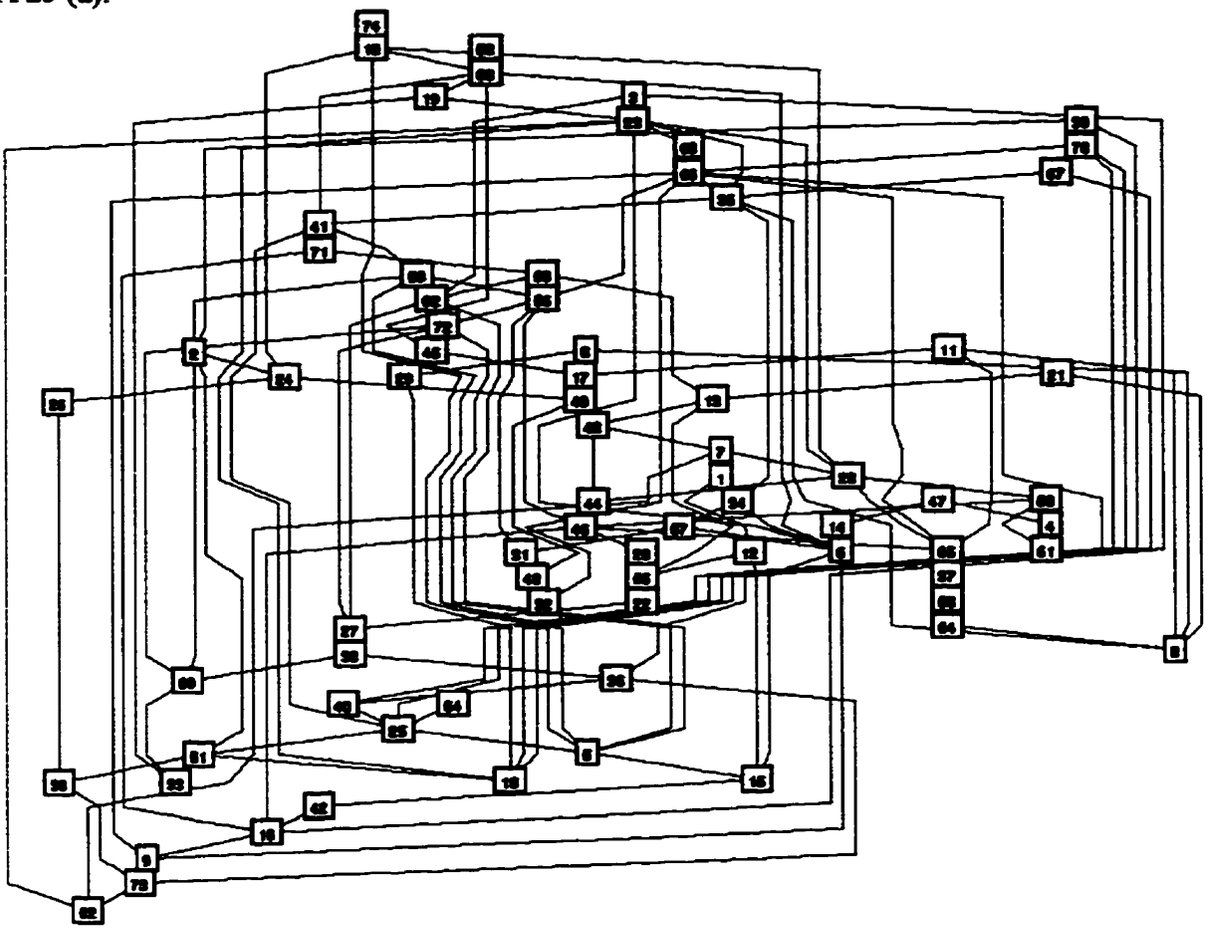
A 24 (a).



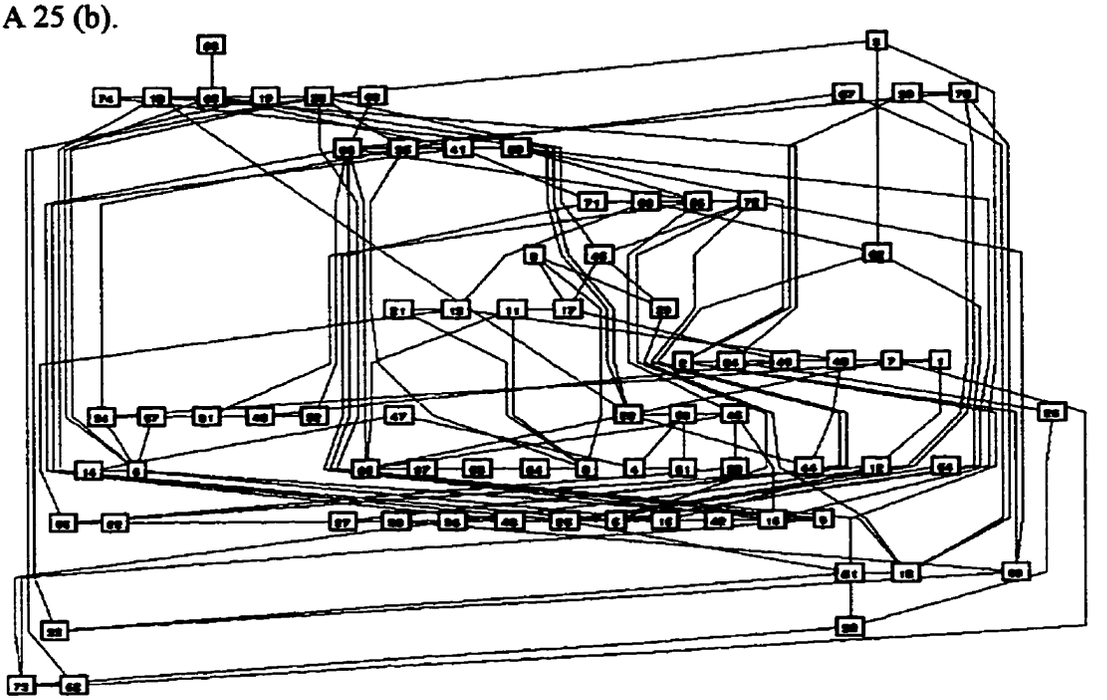
A 24 (b).



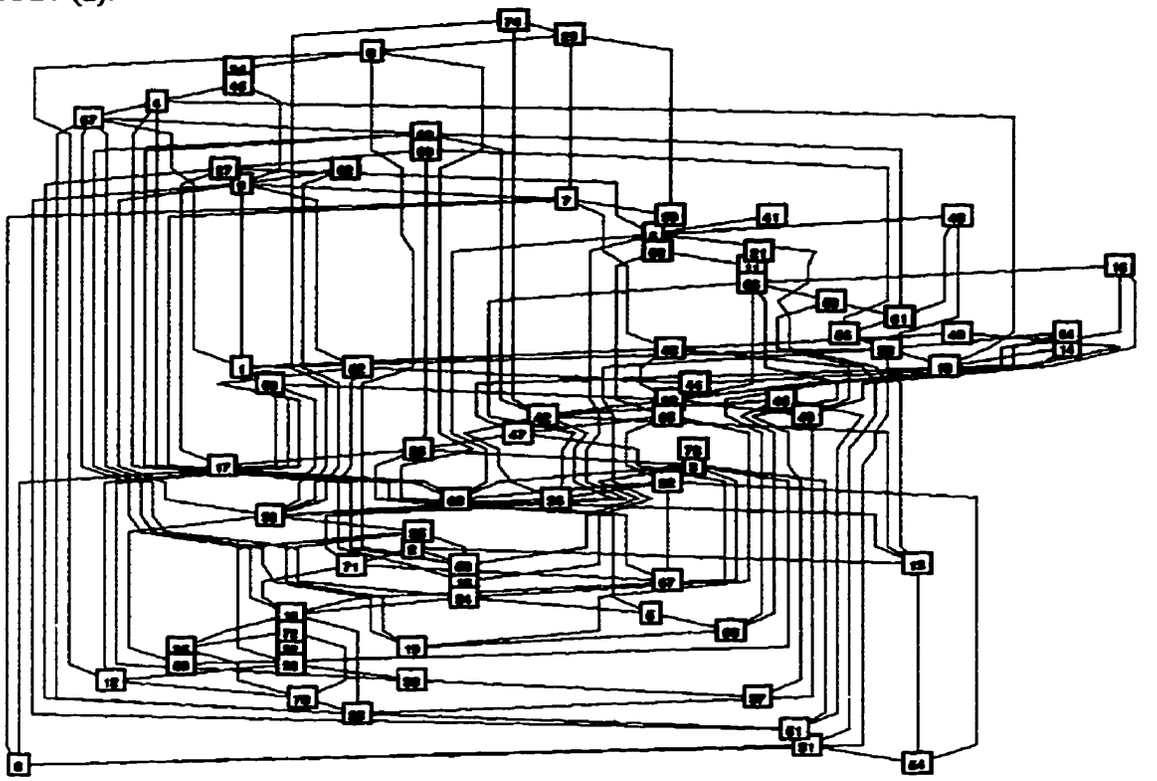
A 25 (a).



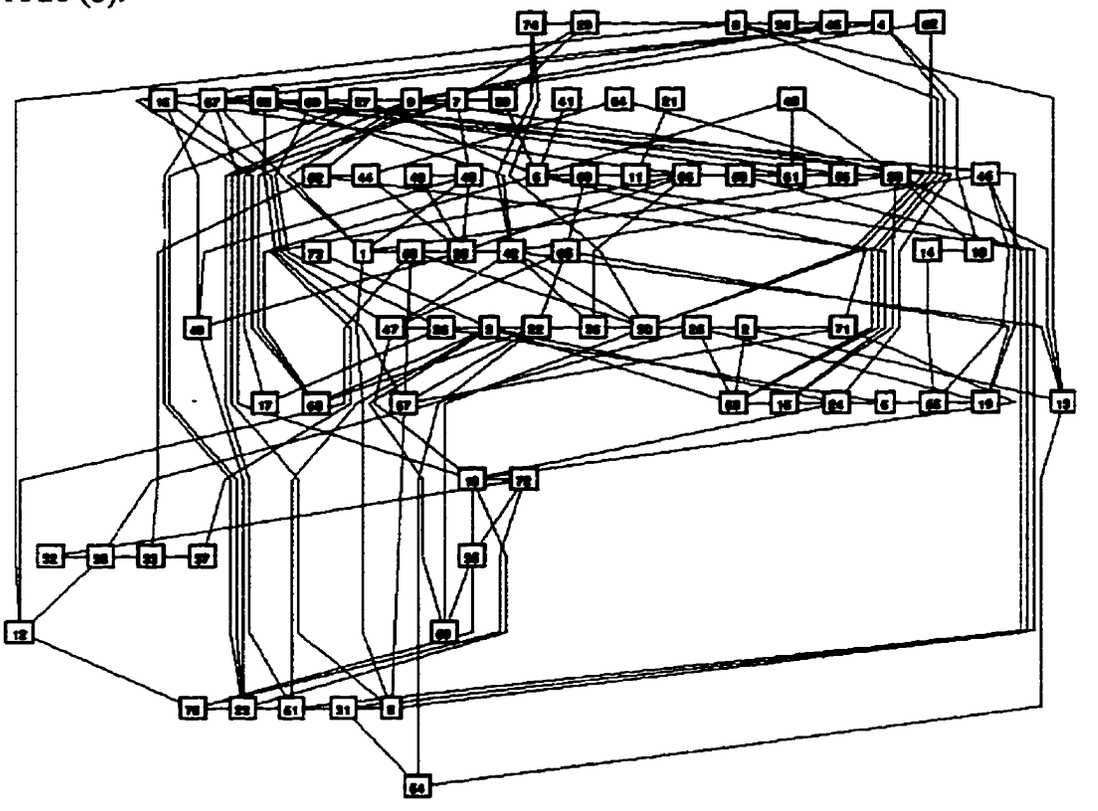
A 25 (b).



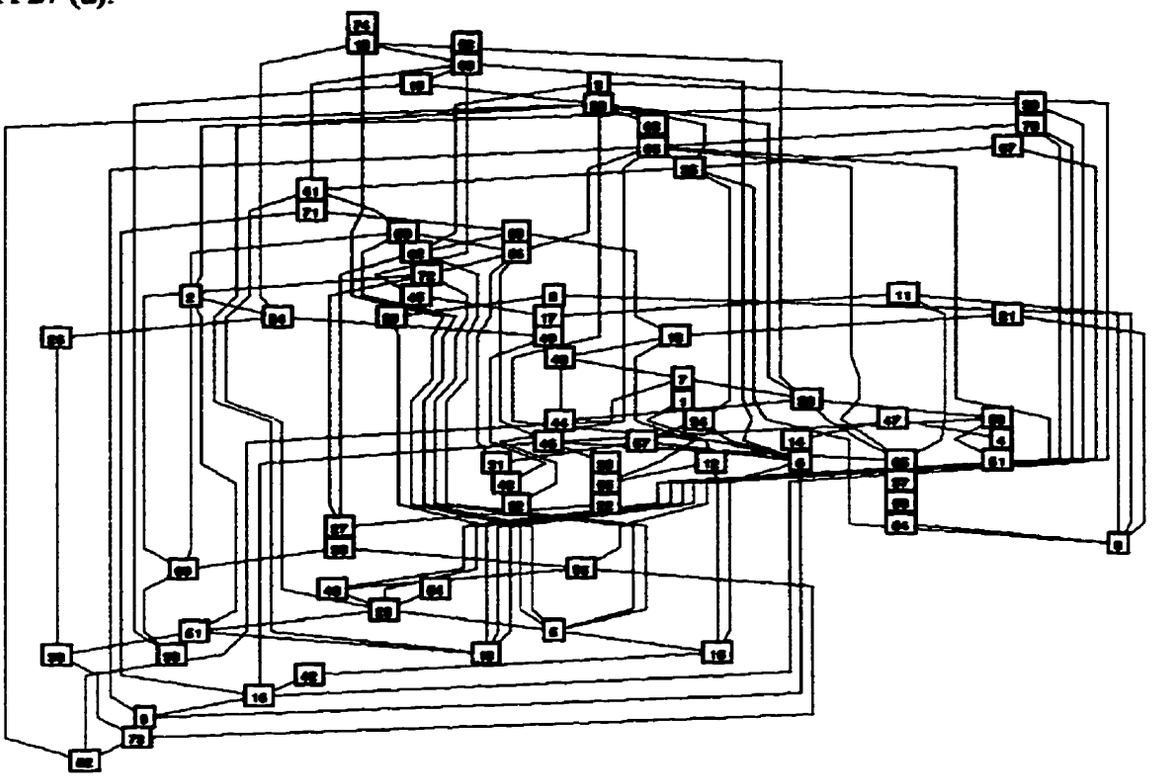
A 26 (a).



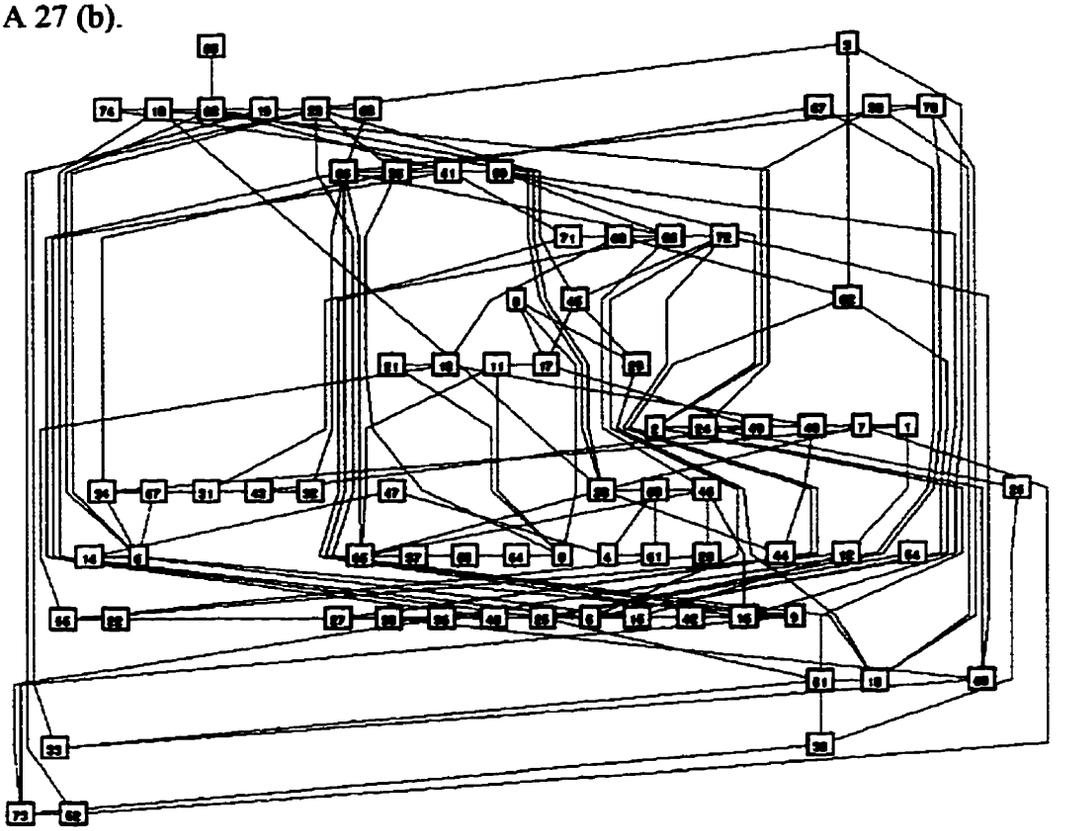
A 26 (b).



A 27 (a).



A 27 (b).



References

[BBL95] P. Bertolazzi, G. Di Battista, and G. Liotta, "Parametric Graph Drawing," *IEEE Trans. on Software Engineering*, vol. 21, no. 8, August 1995.

[BBLM94] P. Bertolazzi, G. Di Battista, G. Liotta, and C. Mannino, "Upward Drawings of Triconnected Digraphs", *Algorithmica*, Springer-Verlag, 1994, pp. 476-497.

[BETT94] G. Di Battista, P. Eades, R. Tamassia and I. G. Tollis, "Algorithms for Drawing Graphs: an Annotated Bibliography", *Computational Geometry Theory and Applications*, vol. 4, 1994, pp. 235-282.

[BT88] G. Di Battista, and R. Tamassia, "Algorithms for Plane Representations of Acyclic Digraphs", *Theoretical Computer Science*, vol 61, 1988, pp.175-198.

[CARP80] M. J. Carpano, "Automatic Display of Hierarchized Graphs for Computer Aided Design Analysis," *IEEE Trans. on Systems, Man, and Cybernetics*, vol. SMC-10, no.11, 1980, pp. 705-715.

[EADES86] P. Eades, B. McKay, and N. Wormald, "On an edge crossing problem," *Proc. 9th Australian Computer Science Conference*, 1986, pp. 327-334.

[FRICK96] Arne Frick, "Upper Bounds on the Number of Hidden Nodes in the Sugiyama Algorithm," *Proceedings of Graph Drawing '96*, 1996.

[GEST47] W. Köhler, *Gestalt Psychology*. New York: New American Library, 1947.

[GKN88] E. R. Gansner, S. C. North, and K. P. Vo, "DAG - A Program that Draws Directed Graphs," *Software Practice and Experience*, vol. 18, no. 11, 1988, pp. 1047-1062.

[GKNV93] E.R. Gansner, E. Koutsofios, S. North, K. Vo, "A Technique for Drawing Directed Graphs", *IEEE Transactions on Software Engineering*, Vol. 19, No. 3, March 1993.

[GRAY91] Peter Gray, *Psychology*, Worth Publishers Inc., 1991.

[HAR69] F. Harary, *Graph Theory*, Addison-Wesley, Reading, MA, 1969.

[HEEST83] A. R. G. Heesterman, *Matrices and Simplex Algorithms*, Boston: D. Reidel Publishing, 1983.

[JOHNS96] David S. Johnson, "A Theoretician's Guide to the Experimental Analysis of Algorithms," AT&T Research, April 30, 1996.

[KOS94] C. Kosak, J. Marks, and S. Shieber, "Automating the Layout of Network Diagrams with Specified Visual Organization," *IEEE Trans. on Systems, Man, and Cybernetics*, vol. 24, no. 3, March 1994.

[MRH91] Eli B. Messinger, Lawrence A. Rowe, and Robert R. Henry, "A Divide-and-Conquer Algorithm for the Automatic Layout of Large Directed Graphs," *IEEE Transactions on Syst. Man. Cybern.* vol. 21, no. 1, January / February 1991.

[NORTH93] S. C. North, "Drawing Ranked Digraphs with Recursive Clusters," AT&T Bell Laboratories, Nov. 3, 1993.

[STT81] K. Sugiyama, S. Tagawa, and M. Toda, *Methods for Visual Understanding of Hierarchical Systems*, IEEE Trans. Systems, Man, and Cybernetics, vol SMC-11, no.2, 1981, pp 109 - 125.

[WARF77] J. Warfield, "Crossing theory and hierarchy mappings," *IEEE Trans. Syst. Man. Cybern.* vol. SMC-7 no. 7, July 1977, pp. 505-523.