Automated Visualization of Process Diagrams*

Janet M. Six and Ioannis G. Tollis

CAD & Visualization Lab
Department of Computer Science
The University of Texas at Dallas
P.O. Box 830688 EC31
Richardson, TX 75083
Fax: (972)883-2349
{janet,tollis}@utdallas.edu

Abstract. In this paper, we explore the problem of producing process diagrams and introduce a linear time technique for creating them. Each edge has at most 3 bends and portions of the edge routing have optimal height. While developing a solution, we explore the subproblems of determining the order of the layers in the diagram, assigning x and y coordinates to nodes, and routing the edges.

1 Introduction

Computer scientists have been using flowcharts for many years as aids for designing, debugging, and documenting algorithmic solutions and system architectures [5,12,15]. These diagrams show the flow of resources, tasks, and time through processes and systems. A *process graph* is a directed graph plus a partitioning of the nodes into groups. Like the structures depicted in flowcharts, the directed graph in a process graph also represents flow through a process or system. The node partition groups can represent any type of classification: e.g., program modules, system components, periods of time, geographic or geometric regions, or social groups. For example, the set of courses required for a degree, their prerequisites, and a grouping of the courses by level can be modeled with a process graph. A visualization of this structure could be quite useful for students as they form their degree plans. A *process diagram* is a drawing of a process graph. See Figure 1 for an example process diagram of the required courses and their prerequisites in the Undergraduate Computer Science Program at the University of Texas at Dallas. The groupings represent the Freshman, Sophomore, Junior, and Senior levels in the program. This process diagram was produced automatically by an implementation of our process diagramming technique.

A process diagram could also be used to model a project management scenario: (a) the nodes and edges could represent tasks and prerequisite constraints

* The research was supported in part by the Texas Advanced Research Program under grant number 009741-040 and a stipend from the Provost's Office at the University of Texas at Dallas.

P. Mutzel, M. Jünger, and S. Leipert (Eds.): GD 2001, LNCS 2265, pp. 45–59, 2002.
© Springer-Verlag Berlin Heidelberg 2002

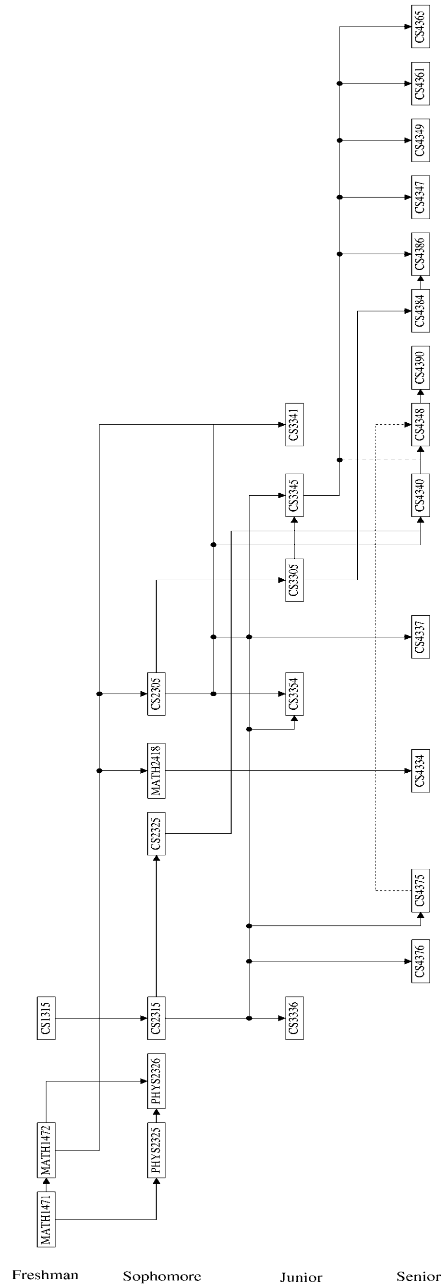

Fig. 1. Process Diagram of the University of Texas at Dallas Undergraduate Computer Science Program requirements where the layers represent the Freshman, Sophomore, Junior, and Senior levels.

and (b) the groups represent the technical and management teams working on the project. Alternatively, (a) the nodes could represent milestones and (b) the groups represent periods of time in which those milestones should be reached. A drawing of a project and its performance with respect to tasks or milestones would be helpful to a project manager.

Define a *layer* to be a horizontal line in the process diagram. Nodes which belong to the same node partition in a process graph are placed on the same layer in a process diagram. Define an *intralayer edge* to be incident to two nodes which are members of the same layer. Likewise, an *interlayer edge* is incident to two nodes which are not members of the same layer.

Since process diagramming is inspired by flowcharting, we present a technique which produces drawings that include the traditional characteristics of flowcharts [5,12,15]. One property of a good process diagram is that the flow through the elements of the graph should start in the top-left corner of the drawing and proceed to the bottom-right corner. Also, the edges should be drawn in an orthogonal manner to promote clarity. The number of bends, number of edge crossings, and edge length should be minimum. Also, nodes of a partition group must not appear on a layer with nodes of another group. The order of the layers can be specified by the user or the technique should order the layers such that the interlayer edges are directed downwards in the resulting process diagram.

Unfortunately, we cannot achieve all of the above properties and conventions simultaneously. For example, it is well known that the minimization of edge bends and crossings often contradict each other [2]. In addition, the problems of minimizing bends, crossings, and edge length are NP-Complete [2]. Therefore, we relax our requirements and strive to produce process diagrams with a low number of bends, crossings, and short edge length. Furthermore, process graphs often contain cycles which include interlayer edges and therefore it is not possible to draw all their edges in a downward direction. Since we cannot guarantee to produce a process diagram with all interlayer edges directed downward, we strive to find a process diagram with the minimum number of upward edges. We also provide a facility to the user to effectively order the layers.

Previously developed techniques do not meet the requirements of process diagramming. Hierarchical graph drawing techniques [2,20] are not suitable to produce process diagrams since the user has no control over the assignment of nodes to layers, no intralayer edges are allowed, the approach is not designed to show rightward flow, and the edges are not orthogonal. Sander presents an algorithm in [18] that draws edges in an orthogonal manner, but this technique does not solve the other problems of process diagramming. Orthogonal graph drawing techniques [2,16] cannot be used to produce process diagrams since they do not take edge directions into account and also that the user has no control over the assignment of nodes to layers.

Knuth presented a technique for automatically drawing flowcharts in [12]. In this algorithm, all nodes are placed in a single column and all edges are drawn to the right of this column. This technique does not allow for a user-based node partitioning. Also, this algorithm was designed to conserve memory

by not requiring the entire data structure to be in memory at one time. Allowing a graph drawing algorithm to analyze an entire structure at once can lead to a more readable drawing.

Process diagrams can be created manually with a tool such as [17], but the effort required of the user can be extensive. It is of benefit to provide a fully automatic tool so that a user does not have to spend a significant amount of time and effort to produce the process diagram.

The GRADE tool [7,11] offers manual, semi-automatic, and automatic avenues for the creation of process diagrams. The tool has the feature of allocating nodes to "lanes" and placing nodes of a lane within a horizontal or vertical box. These lanes can be used to handle the node partitions in a process graph. However, the automatic layout technique described in [11] first uniformly places the nodes in the layout and then uses the *barycenter* method to finalize the node positions. The barycenter method for graph layout iteratively places each node in the midst of its neighbors. From the layout description given in [11], the worst case time requirement is unclear. Furthermore, for every $n > 1$, there exists a graph with n nodes such that the barycenter method requires exponential area [2]. Also, the edge routing is not described and therefore we do not know of an upper bound for the number of bends per edge.

Showbiz [22] is a process flow modeling tool. A facility is given to the user to visualize a flow with the additional quality that each node is placed in a lane that represents an attribute of that node. The x coordinates of the nodes are determined with the first phase of the Sugiyama hierarchical algorithm [20] and the y coordinate is given based on the location of the lane. Sometimes nodes can overlap and then the lane is widened so that those nodes appear one above the other. This layout technique visualizes rightward, but not downward flow. Also, some non-orthogonal edges are used in the edge routing. This technique is not sufficient to solve the process diagramming problem.

In this paper, we present a systematic $O(m)$ time approach for producing process diagrams, where m is the number of edges. These diagrams have at most 3 bends per edge and portions of the edge routing have optimal height.

2 A Linear Time Technique for Producing Process Diagrams and Determining the Order of the Layers

The problem of creating a process diagram can be divided into the following tasks:

1. Determine an order of the layers as they should appear from top to bottom.
2. Assign x and y coordinates to each node.
3. Route the edges.

We will now look at each task in detail and present algorithms to solve them. Then we will put these solutions together in Section 5 and present a linear time algorithm for producing process diagrams.

Let $G = (V, E, P)$ be a process graph, where V is the set of nodes, E the set of edges, and P the node partitioning. Our first task is to determine an ordering of the layers, corresponding to each partition, as they should appear from top to bottom in the process diagram Γ. We present a technique which finds an ordering with the following properties:

1. The layers which contain source nodes appear at the top of the drawing.
2. The layers which contain sink nodes, but no source nodes, appear at the bottom of the drawing. If there is a layer which contains both source and sink nodes, that layer is treated as one which contains source nodes and appears at the top of the drawing.
3. The layers are ordered such that most interlayer edges of the input graph are directed downwards in the resulting process diagram.

Given a process graph $G = (V, E, P)$, we find an ordering of the layers with the above properties by first creating a *layer graph*, $G' = (V', E')$. The nodes in V' represent the node partitions in P. The set of edges in E' is determined in the following manner:

For each pair of nodes in V', L_i and L_j

1. Examine the set E_{ij} of edges in E which have one incident node in L_i and L_j.
2. If neither L_i nor L_j contain a source or sink node then add a dominant type edge to E'. (If the majority of edges in E_{ij} are directed from L_i to L_j, we say that (L_i, L_j) is the *dominant type edge*. Otherwise, (L_j, L_i) is the dominant type edge.)
3. If L_i (or L_j) contains a source node then add (L_i, L_j) (or (L_j, L_i)) to E'.
4. If L_i (or L_j) contains a sink node then add (L_j, L_i) (or (L_i, L_j)) to E'.
5. If both L_i and L_j contain source (or sink) nodes then add a dominant type edge to E', as described in Step 2.

If a node represents a layer which contains both source and sink nodes, we treat it as a layer with source nodes. Not only does this process assure that no multiedges exist in G', it also prevents the inclusion of cycles of length two (*two-cycles*).

If G' is a directed acyclic graph, then we can apply a *topological sort* [1] to find a good layer order. Often times layer graphs contain cycles and therefore it is necessary to employ a more advanced method to obtain the order. We can transform a graph with cycles into an acyclic graph by reversing a set of edges. In order to produce a process diagram with the maximum number of interlayer edges directed downward, we want to reverse the minimum number of edges in the layer graph. This is equivalent to the NP-Complete *feedback arc set problem* [2,6,10] in which a minimum cardinality set of edges is removed in order to make the graph acyclic. In [3], a $O(m)$ time heuristic algorithm for reversing a small number of edges in order to make a graph acyclic is presented. The algorithm, *Greedy-Cycle-Removal*, orders the nodes of the input graph such that the number

of edges going from a node later in the order to a node earlier in the order is small. These edges are called *backward* and form a set which can be reversed in order to produce an acyclic graph. The other edges are *forward*. The input graph to this algorithm is assumed to be connected, however it can easily be extended to handle disconnected graphs by performing Greedy-Cycle-Removal on each connected component successively.

There are properties of the resulting sequence S which are guaranteed if the input graph does not contain any two-cycles. Since layer graphs do not contain two-cycles, we can also guarantee the same performance.

Theorem 1. *[3] Suppose that G is a connected directed graph with n nodes and m edges, and no two-cycles. Then Greedy-Cycle-Removal computes a node ordering S of G, with at most $m/2 - n/6$ backward edges.*

This result is strengthened for graphs whose underlying undirected graph contains no nodes with degree greater than three. In those cases, Greedy-Cycle-Removal computes a node ordering of G with at most $m/3$ backward edges [2, 4]. In our case, we are applying Greedy-Cycle-Removal to G', which is typically a small-sized graph. In fact, in some cases the graph may be so small that an exponential search to find the best set of edges to reverse can be conducted in a reasonable amount of time.

These theorems show that Greedy-Cycle-Removal has a guaranteed performance much better than the worst case of another technique for making a graph acyclic which performs a DFS on a given directed graph and reverses all the back edges. In fact, in the worst case, $m-n-1$ edges may be reversed. In addition to having a guaranteed performance, Greedy-Cycle-Removal can be implemented to run in linear time and space [3].

We now present an $O(m)$ time technique for determining the order of layers for a process diagram which has been inspired by the algorithm in [3]. In addition to finding an order of the layers, the nodes of the input graph are also given a y coordinate according to the order which is found.

Algorithm 1 *DetermineOrderOfLayers*
Input: A process graph, $G = (V, E, P)$.
Output: An ordering of the layers.

1. Create a layer graph G' from process graph G as described above.
2. Initialize S_l and S_r to be empty lists.
3. While G' is not empty do
   a) While G' contains a source u, remove u and append it to S_l.
   b) While G' contains a sink v, remove v and prepend it S_r.
   c) If G' is not empty then choose a node w, such that the value $outdegree(w) - indegree(w)$ is maximum, remove w from G' and append it to S_l.
4. Concatenate S_l with S_r to form S.
5. Output S.

Applying Algorithm 1 to the process graph shown in Figure 1, the resulting layer graph would have 4 nodes (Freshman, Sophomore, Junior, and Senior) and 3 edges ((Freshman, Sophomore), (Sophomore, Junior), (Junior, Senior)). In Steps 4 and 5 of the algorithm, we would find S to be (Freshman, Sophomore, Junior, Senior).

3 Determining the x Coordinates of the Nodes

We divide the problem of assigning x coordinates to nodes into two subtasks: first for each layer, we create groups of nodes. Secondly, we order those groups in the layer such that most intralayer edges are rightward in the process diagram.

First, we discuss Algorithm *PartialOrdering*. To increase readability we wish to place nodes which are members of a long path consecutively on a layer. A node v is *reachable* from u if there exists some path from u to v. Nodes which are reachable from node u form the *reachability group* of u. We find the reachability groups in the subgraph $G_i = (V_i, E_i)$ induced by the nodes assigned to Layer L_i by employing a DFS. If G_i is not strongly connected, there will be multiple reachability groups. These groups can contain nodes of different strongly connected components, but not necessarily be the same as the connected components of the underlying undirected graph.

In order to find a good set of reachability groups, we will always start the DFS at a node of smallest indegree. If the DFS algorithm must choose a node to start an additional tree in the depth first spanning forest, it will also start at an unvisited node of lowest indegree. The result of these steps, in addition to the depth first spanning forest, is a set of groups each of which contains members of a tree from the forest. The nodes in each group are in the same order as visited in the DFS, thus providing a partial ordering. This process requires $O(m)$ time since it is a variant of DFS. Figure 2 shows the application of Algorithm PartialOrdering on the subgraph of the Sophomore layer in Figure 1. The left side of the figure shows the elements of the subgraph while the right side shows the grouping of the elements into reachability groups.

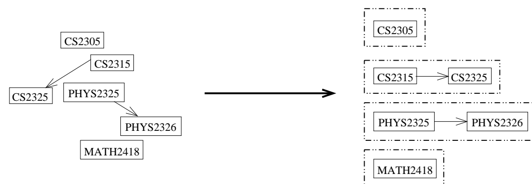

Fig. 2. The application of Algorithm PartialOrdering on the subgraph of the Sophomore layer in Figure 1.

The second subtask in determining the x coordinates is to order the reachability groups found with Algorithm PartialOrdering. A *weak dominance drawing*

has the property that for any two vertices u and v, if there is a directed path from u to v then v is placed below and/or to the right of u in the drawing. This is a variant of the *dominance drawing* technique for planar *st*-graphs discussed in [2]. In a dominance drawing, a node v is given x and y coordinates greater than or equal to that of another node u if and only if there exists a directed path from u to v. Weak dominance drawing captures the down-and-to-the-right flow we want to see in process diagrams. For planar acyclic graphs, dominance drawing algorithms presented in [2] could be used to create a process diagram. However, the technique presented in this paper works on all process graphs.

We next discuss Algorithm *DetermineXCoordinates* which will order the groups. We place the reachability groups of all layers in the order they are found by Algorithm PartialOrdering into a queue Q. We set $currentSourceColumn$ to be the leftmost column. We process the members of Q and place the groups which contain a source node. At each iteration, we dequeue a group α: if it contains a source node, then we place the first node of α in $currentSourceColumn$ and the remaining nodes in the next $|\alpha| - 1$ columns. We insert columns as necessary to avoid nodes of the same layer being assigned to the same column. After the nodes of α are placed, we increment $currentSourceColumn$ by $|\alpha|$. If α does not contain a source node, we enqueue it back into Q.

After all the groups with source nodes have been placed, we process the members of Q again and place the remaining groups. We dequeue a group α and determine if it contains a node which is incident to a node which has been placed. Due to the construction of Q and the expected down-and-to-the-right flow in the process graph, it is likely for a neighbor of a node in α to already be placed. If this is the case, we find the rightmost placed neighbor u of a node in α. If there is no obstacle in the column of u between the row of u and the row of α and a node has not already been placed in the column of u and the row of α, then place α starting in column of u. If there is an obstacle in the column of u between the row of u and the row of α and a node has not already been placed in the column of $u + 1$ and the row of α, then place α starting in column of $u + 1$. Columns are inserted as necessary to avoid the placement of groups being intermingled. Otherwise, if we have been unable to place α with one of the above methods, then we place α starting in the rightmost column being used for the layer of α. Since, we process the queue at most twice, this algorithm requires $O(m)$ time. The columns in the process diagram are labeled in ascending order from left to right and denote an x coordinate. In Figure 3, we show the application of Algorithm DetermineXCoordinates on the reachability groups shown in Figure 2. The reachability group with PHYS2325 and PHYS2326 is placed to the left of the reachability group with CS2315 and CS2325 since a relative of PHYS2325, MATH1471, was placed to the left of a relative of CS2315, CS1315, on a higher layer. MATH2418 and CS2305 are placed in columns to the right due to obstacles. The relatives of these 6 nodes which belong to lower layers will be placed below and to the right of these nodes on lower layers. This type of placement facilitates the down-and-to-the-right flow present in the process diagrams created by the approach in this paper.

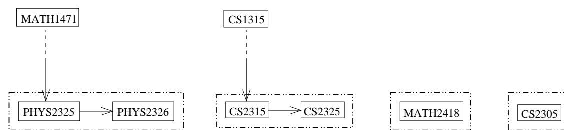

Fig. 3. The application of Algorithm DetermineXCoordinates on the reachability groups in Figure 2.

4 Routing the Edges and Determining Final Coordinates

The task of routing the edges in a process diagram is related to the VLSI design problem *Channel Width Minimization within the Jog-Free Manhattan Model* (CWM) [14]. In CWM, the terminals (or nodes) are given fixed placements on two horizontal lines and the nets (or edges) are routed in the area between the two lines. This area is called the *channel*. Horizontal wire segments occupy *tracks* and the number of tracks used is referred to as the *width* of the channel. The CWM is defined as follows: given two sets of terminals which have been placed on two different horizontal lines and a set of nets, find a routing such that each net has at most one horizontal segment and the channel is of minimum width [14]. The CWM is NP-Hard [13]. Define a *horizontal constraint* to exist between two nets if they would overlap if placed in the same track. Define a *vertical constraint* to exist between two nets if one terminal of each net resides in the same column. The general CWM is NP-Hard. However an optimal solution can be found in linear time for those instances of CWM which have no vertical constraints [14].

Theorem 2. *[14] Given an instance of CWM which has no vertical constraints, there does exist an algorithm which finds a channel of minimum width. Furthermore, this solution can be found in $O(m)$ time, where m is the number of nets which need to be routed.*

Hashimoto and Stevens present the *left-edge algorithm* in [9] as a solution to CWM. The main loop of the algorithm scans the channel from left to right and places each net in a track such that no two nets overlap. It packs the horizontal segments of nets as tightly as possible. We can apply the left-edge algorithm to our edge routing problem by treating the area between two layers as a channel and assigning a set of edges to each channel. In order to encourage the appearance of a down-and-to-the-right flow in process diagrams, we place the attachment points of incoming edges on the left and top sides of their target nodes. Likewise, outgoing edges emanate from the bottom and right sides of their source nodes. If we hold these properties strictly, we would add edge length or bends to process diagrams while routing edges from lower to higher layers. To avoid these problems, we route these edges emanating from the top of their source nodes. We also route outgoing edges from the top of the source node if they are directed rightward to a node on the same layer. If a node u has an outgoing edge to a node on a higher layer or directed rightward to a node on the same layer, we

mark the top of u as *reserved*. This distinction prevents incoming and outgoing edges being attached at the same point on a node.

There are ten types of edge routing in process diagrams. See Figure 4. Assume u to be on Layer L_i. Node v is on Layer L_{i+1} or lower if it appear below u or on Layer L_{i-1} or higher if it appear above u. Routings (a) and (b) are used for downward and rightward edges for which the source and target nodes share a row or column. Routing types (c), (d), and (e) are for edges with their source and target on the same layer. Types (f), (g), (h), and (i) are for edges with their source on a higher layer than the target. Type (j) is for edges with their source on a lower level than the target. These edge routings have at most three bends per edge. As is shown in Figure 4, downward edges can be routed with routing types (a), (f), (g), (h), or (i). We can reduce the number of bends and possibly the area if we use routing type (a), but this routing is not always feasible. Therefore, we try to route a downward edge with first a type (a), then (f), (g), (h), and (i) type routings. Similarly, we try to route a rightward edge between nodes of the same layer with edge routing type (b) then (c) and (d). These sequences are used in order to reduce the number of bends, edge length, and area of process diagrams.

Let Channel $L_{i-1}L_i$ represent the area above Layer L_i. Likewise, let Channel L_iL_{i+1} represent the area below Layer L_i. We will assign horizontal segments of type (c), (d), (e), (h), (i), and (j) routings to some channel and then place them with the left-edge algorithm. In order to use the left-edge algorithm, we need to guarantee that no vertical constraints exist. In our edge routing problem, we have a vertical constraint if a Layer L_i node u in column x_u is the source of an edge routed with type (e), (h), or (i) and a Layer L_{i+1} node v is in column x_u and is the source of an edge routed with type (c), (d), or (j) or is the target of an edge routed with type (h) routing. We can avoid a type (e, h, i) / (h) vertical constraint, by routing the edge directed toward v with a type (i) routing. We can avoid type (e, h, i) / (c, d, j) vertical constraints by using an edge routing with more bends, however we can maintain our three bends per edge property by moving v to a new column x_v+1 and still using types (c), (d), or (e) routings. In order to avoid adding many columns, we add a new column after x_v at most one time. This new column can resolve multiple vertical constraints.

We will now examine each edge routing type in detail. If the source and target of a downward or rightward edge $e = (u, v)$ reside in the same row or column, we attempt to route e with a type (a) or (b) routing. Let x_u and y_u be the x and y coordinates of node u. Let x_u (y_u)to denote the column (row) of u. If e is downward, the top of v is not reserved, and there are no obstacles in the process diagram from (x_u, y_u) to (x_v, y_v), then we route e with a straight edge from (x_u, y_u) to (x_v, y_v). Likewise, if e is rightward and there are no obstacles in the process diagram from (x_u, y_u) to (x_v, y_v), then we route e with a straight edge from (x_u, y_u) to (x_v, y_v). If obstacles are encountered, we will route the edge with another routing type.

If we are unable to route a rightward edge $e = (u, v)$ with a type (b) routing, we will route the edge with a type (c) or (d) routing. If the top of v is not

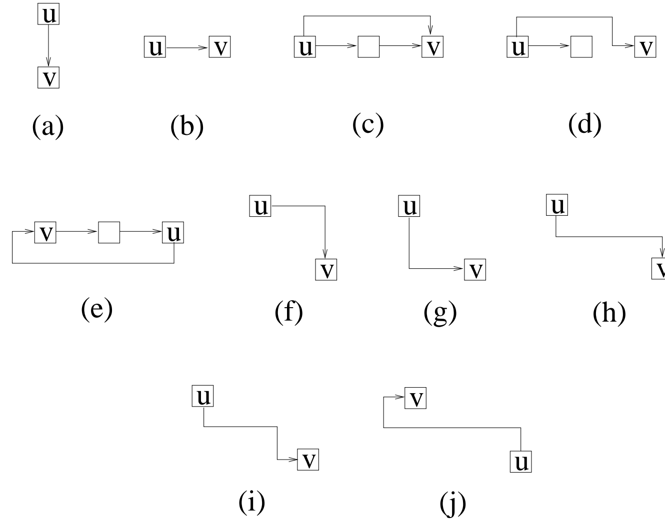

Fig. 4. Ten possible edge routings for the edge (u, v).

reserved, then we can route e with type (c), else we route with type (d). In a both types (c) and (d) routings, we place the first bend in the column of u. In a type (c) routing, we place the second bend in the column of v. We insert a new column, $x_v - 1$, for the placement of the second bend in a type (d) routing. The third bend will be placed at $(x_v - 1, y_v)$. Since we want edges routed with type (c) or (d) to be drawn above Layer L_i, we place the first horizontal segment into Channel $L_{i-1}L_i$. We will then use the left-edge algorithm to place this segment and thus determine the y coordinates of the first two bends.

Type (e) routing is for a leftward edge $e = (u, v)$ with source and target nodes in the same layer L_i and is very similar to type (d) routing. A new column is introduced at $x_v - 1$ and the first two bends placed in the columns x_u and $x_v - 1$. The third bend is placed at $(x_v - 1, y_v)$. Since we want these edges to appear below layer L_i, we assign edge e to Channel L_iL_{i+1} and use the left-edge algorithm to place the first horizontal segment of e.

Routing types (f), (g), (h), and (i) are used to route downward edges. Both types (f) and (g) attempt to route a downward edge $e = (u, v)$ with a one bend edge. Routing type (f) is used if the top of v is not reserved, u does not have an outgoing edge routed with type (b), and there are no obstacles in the process diagram from (x_u, y_u) to (x_v, y_u) and from (x_v, y_u) to (x_v, y_v). Alternatively, type (g) can be used if there are no obstacles in the process diagram from (x_u, y_u) to (x_u, y_v) and from (x_u, y_v) to (x_v, y_v). If neither type (f) nor (g) can be used, then we will route with type (h) or (i). The first bend in a (h) or (i) type routing will be placed in the column of u. If the top of v is not being used and a type (h) routing will not create a vertical constraint, then we will use a type (h) routing and place the second bend in the column of v. Else, we insert a new column

$x_v - 1$ and place the second bend in the new column. The third bend is placed at $(x_v - 1, y_v)$.

Define a *proxy edge* to be an edge which represents a group of edges. If we use a proxy edge to represent the k outgoing downward edges of u, then we can route those k edges with one horizontal edge segment in the process diagram as opposed to using k horizontal edge segments. Multiple edges are allowed to come into the top of u if there is no edge directed from u to a node on a higher layer or there is no incoming straight edge to the top of u. Multiple edges can also come into the left side of a node via a single horizontal edge segment. We avoid ambiguity by allowing an edge segment in the process diagram to have edges either branching out or merging in on a line segment, but not both. This difference can be further highlighted by using dots to mark the points where edges branch out and not using dots to mark the points where edges merge together. To reduce the height and edge length of process diagrams, we use a proxy edge e' to represent all outgoing edges of u routed with type (h) or (i). Edge e' is assigned to Channel L_iL_{i+1}. After the left-edge algorithm is applied, we place the first horizontal segment of all the edges represented by proxy edge e' in the track assigned to e'.

Upward edges are routed with type (j) from Figure 4. For an upward edge $e = (u, v)$, we place the first bend in the column of u and insert a new column, $x_v - 1$, for the second bend. The third bend is placed at $(x_v - 1, y_v)$. The first horizontal segment of e is assigned to Channel $L_{i-1}L_i$.

We now give an algorithm for the routing of edges in process diagrams.

Algorithm 2 *RouteEdges*
Input: A process graph, $G = (V, E, P)$ and
A set of (x, y) coordinates for the nodes in V.
Output: A routing for all edges in E.

1. Resolve any vertical constraints by adding new columns as described in the previous text.
2. Attempt to route downward and rightward edges with source and target nodes on the same layer with type (a) and (b) routings.
3. Route rightward edges not routed in Step 2 with a type (c) or (d) routing.
4. Route leftward edges with source and target nodes on the same level with type (e) routings.
5. Route downward edges not routed with Step 2 with type (f), (g), (h), or (i) routings.
6. Route upward edges with type (j) routings.
7. For each channel, L_iL_{i+1},
   a) Set *currentRow* to be the row of Layer L_i.
   b) Set *currentColumn* to be the leftmost column.
   c) Set *maxColumn* to the rightmost column.
   d) While edges in L_iL_{i+1} not routed and $currentColumn \neq maxColumn$
      i. Set *newRow* to be a newly inserted row below *currentRow*.
      ii. Set *currentRow* to *newRow*.

iii. Find the next column which contains an edge to be routed, place that edge in *currentRow*, and set *currentColumn* to the column after the column of the rightmost bend of the newly placed edge. If no edge can be routed on the current row, then set *currentColumn* to *maxColumn*.

Resolving the vertical constraints in Step 1 requires $O(n)$ time. Assigning a routing type to each edge in Steps 2 - 6 requires $O(m)$ time. The remaining steps are a variant of the left-edge algorithm [9]. From Theorem 2, we know that these steps require $O(m)$ time. The left-edge algorithm can be implemented to run in linear time if any available track can be used to route an edge as opposed to using the available track which is closest to the top of the channel [8].

Due to Theorem 2, we know that Algorithm 2 will find channels of minimum width given the assignment of edge segments to channels. In order to prove this approach finds an optimal solution [14], let m_j be a edge that is assigned to the track t_j which is furthest from the first line of nodes α. For each track, t_k, closer to α, there must be some edge, m_k, which has a horizontal constraint with m_j, otherwise m_j would be placed on track t_k due to the construction on the algorithm and the stipulation that there are no vertical constraints.

Define the *closed density* δ of an instance of CWM to be equal to $\max_x |E_x|$, where the columns of the channel are labeled in ascending order from left to right, x is a number, and E_x is the set of edges whose routings must pass the column labeled x under the constraints of the jog-free manhattan model. The maximum channel width found with the left-edge algorithm is the closed density δ. Moreover, we know that δ is the minimum channel width necessary to allow a legal routing. Therefore, the left-edge algorithm produces an optimal result, with respect to channel width, when given an instance of CWM without vertical constraints.

5 Creating a Process Diagram in Linear Time

We put the previous algorithms together in this section and present a $O(m)$ time technique for creating process diagrams.

Algorithm 3 *CreateProcessDiagram*
Input: A process graph $G = (V, E, P)$, and
An ordering of the layers (optional).
Output: A process diagram of G, Γ, with the conventions and properties discussed in the earlier text.

1. If the ordering of the layers is not given, then determine the ordering of the layers with Algorithm DetermineOrderOfLayers.
2. Determine the partial orderings of the nodes within each layer with Algorithm PartialOrdering.

3. Determine a x coordinate for each node with Algorithm DetermineXCoordinates.
4. Route the edges with Algorithm RouteEdges.

Algorithm 3 requires $O(m)$ time. Each edge has at most three bends in Γ and the height of the channels are minimum. This optimality does not guarantee that our drawing is of minimum height since it is possible that edges need to be assigned to other channels. The height of the drawing is bound by the number of partitions in P plus n (to represent the type (h) and (i) edge routings) plus the number of rightward, leftward, and upward edges. The width of the drawing is bound by the two times the number of nodes plus the number of edges. A sample process diagram as produced by an implementation of Algorithm 3 is shown in Figure 1.

We have implemented Algorithm 3 in C++ and include some preliminary results below. The first two process graphs are course flow charts for The University of Texas at Dallas. The third graph is a modified flowchart. The fourth and fifth process graphs are not acyclic. The criteria included for these process diagrams include the size of the process graph (number of nodes, edges, and partitions), total number of edge bends, maximum number of bends per edge, and number of edge crossings. The number of edges that are drawn with a down-and-to-the-right flow is also shown. This criterion shows how well our process diagrams capture the down-and-to-the-right flow in these process graphs. Finally, we show the number of rows saved with the use of proxy edges.

Symbol	Criterion Represented
β	The number of edge bends
η	The maximum number of bends per edge
χ	The number of edge crossings
ρ	The number of edges with down-and-to-the-right-flow
ϵ	The number of rows saved by using proxy edges

	$\|V\|$	$\|E\|$	$\|P\|$	β	η	χ	ρ	ϵ
Ex. 1	27	34	4	48	3	24	34	13
Ex. 2	35	42	4	59	3	25	41	9
Ex. 3	18	24	9	7	1	0	24	0
Ex. 4	18	25	8	24	3	8	19	0
Ex. 5	70	96	11	113	3	95	88	7

More details of this algorithm and extensions of this technique to handle nodes of special type are discussed in [19].

References

1. T. H. Cormen, C. E. Leiserson and R. L. Rivest, *Introduction to Algorithms*, McGraw-Hill, 1990.
2. G. Di Battista, P. Eades, R. Tamassia and I. G. Tollis, *Graph Drawing: Algorithms for the Visualization of Graphs*, Prentice-Hall, 1999.
3. P. Eades, X. Lin and W. F. Smyth, A Fast and Effective Heuristic for the Feedback Arc Set Problem, *Information Processing Letters*, 47, pp. 319-323, 1993.
4. P. Eades and X. Lin, A New Heuristic for the Feedback Arc Set Problem, *Australian Jrnl. of Combinatorics*, 12, pp. 15-26, 1995.
5. M. V. Farino, *Flowcharting*, Prentice-Hall, 1970.
6. M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, 1979.
7. GRADE User Manual, Infologistik GmbH, 2000. Also available at `http://www.gradetools.com`.
8. U. I. Gupta, D. T. Lee and J. Y.-T. Leung, An Optimal Solution for the Channel Assignment Problem, *IEEE Trans. Computers*, C-28(11), pp.807-810, 1979.
9. A. Hashimoto and J. Stevens, Wire Routing by Optimizing Channel Assignment with Large Apertures, *Proc. 8th ACM/IEEE Design Automation Workshop*, pp. 155-169, 1971.
10. G. Isaak, Tournaments as Feedback Arc Sets, *Electronic Jrnl. of Combinatorics*, 2(1), #R20, 1995. Also available at `http://www.combinatorics.org`.
11. P. Kikusts and P. Rucevskis, Layout Algorithms of Graph-Like Diagrams for GRADE Windows Graphic Editors, *Proc. of GD '95*, LNCS 1027, Springer-Verlag, pp. 361-364, 1996.
12. D. E. Knuth, Computer Drawn Flowcharts, *Comm. of the ACM*, 6(9), 1963.
13. A. S. La Paugh, *Algorithms for Integrated Circuit Layout: An Analytic Approach*, PhD Thesis, Massachusettes, 1980.
14. T. Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout*, John Wiley and Sons, 1990.
15. J. Martin and C. McClure, *Diagramming Techniques for Analysts and Programmers*, Prentice-Hall, 1985.
16. A. Papakostas, *Information Visualization: Orthogonal Drawings of Graphs*, Ph.D. Thesis, The University of Texas at Dallas, 1996.
17. K. Ryall, J. Marks, and S. Shieber, An Interactive System for Drawing Graphs, *Proc. GD '96*, LNCS 1190, Springer-Verlag, pp. 387-94, 1997.
18. G. Sander, A Fast Heuristic for Hierarchical Manhattan Layout, *Proc. GD '95*, vol 1027 of *LNCS*, pp. 447-458, Springer-Verlag, 1996.
19. J. M. Six, *VisTool: A Tool For Visualizing Graphs*, Ph.D. Thesis, The University of Texas at Dallas, 2000.
20. K. Sugiyama, S. Tagawa and M. Toda, Methods for Visual Understanding of Hierarchical Systems, *IEEE Trans. Systems, Man and Cybernetics*, SMC-11, no. 2, pp. 109-125, 1981.
21. R. Tamassia, On Embedding a Graph in the Grid with the Minimum Number of Bends, *SIAM J. Comput.*, 16, pp. 421-444, 1987.
22. K. Wittenburg and L. Weitzman, Qualitative Visualization of Processes: Attributed Graph Layout and Focusing Techniques, *Proc. GD '96*, LNCS 1190, Springer-Verlag, pp. 401-8, 1997.