METHOD FOR ORTHOGONAL EDGE ROUTING OF DIRECTED LAYERED GRAPHS WITH EDGE CROSSINGS REDUCTION

Jordan Raykov

JDElite Consulting, Boulder, Colorado, USA

ABSTRACT

This paper presents a method for automated orthogonal edge routing of directed layered graphs using the described edge crossings reduction heuristic algorithm. The method assumes the nodes are pre-arranged on a rectangular grid composed of layers across the flow direction and lanes along the flow direction. Both layers and lanes are separated by rectangular areas defined as pipes. Each pipe has associated segment tracks. The edges are represented as orthogonal polylines consisting of line segments and routed along the shortest paths. Each segment is assigned to a pipe and to a segment track in it. The edge crossings reduction uses an iterative algorithm to resolve crossings between segments. Conflicting segments are reassigned to adjacent segment tracks, either by swapping with adjacent segments, or by inserting new tracks and calculating the shortest paths of edges. The algorithm proved to be efficient and was implemented in an interactive graph design tool.

KEYWORDS

Directed Graphs, Orthogonal Edge Routing, Crossings Reduction Algorithms.

1. INTRODUCTION

Graph drawings are commonly used to model or document data flows, complex data structures or processes, system behaviour and management, knowledge representation, and many others. Most often the orthogonal edge drawing is being used as the common graph presentation mode. The main goal of the tools for automated generation of graph drawings is to produce diagrams with uncluttered edge paths around the nodes and with minimal number of crossings between the edges. There are already plenty of techniques for orthogonal drawing of edges. Some known solutions use sophisticated path-finding algorithms [1][3]. Other solutions, based on the layered node layout, introduce abstract virtual nodes on each intermediate layer in search of optimal visibility routes [6][7]. Some other solutions reduce the task to the consideration of acyclic graphs only [2][4]. Most existing solutions are narrowly oriented toward specific classes of diagrams [8]. Those solutions do not cover all cases, especially for large and complex graphs or when it comes to implementations in interactive diagramming tools.

As we know, the reduction of edge crossings is a complex mathematical problem in the computational complexity theory, considered to be an NP-complete decision problem [5]. Most often the NP-complete problems are handled by using heuristic methods. As the graph size grows, the reduction of edge crossings becomes increasingly difficult. When the graph nodes do not have strict positioning constraints, rearranging some nodes may eliminate part of the crossings between edges connected to these nodes. However, in most cases the clarity of the drawing has the highest priority, which restricts the positions of most nodes, often within certain areas of the diagram. Currently existing solutions use heuristic techniques that provide partial crossings minimization. With more complex graphs these proposed techniques usually do not achieve satisfactory results, introducing unpredictable clutter or aesthetically unappealing and difficult to read diagrams.

We begin with some definitions and terms as well as with a brief description of the general approach to reduce edge crossings. We borrow some concepts and terminology from [5] related to the orthogonal drawing conventions for directed graphs, as well as some considerations for edge crossings reduction. Additionally, we present more definitions reflecting the practical aspects of the design.

2. DESCRIPTION

The common definition of a directed graph (digraph) is $G = (V, E)$, consisting of a set V of vertices (nodes) and a set E of edges (connections), where each connection is represented (in our case) as ordered pair (u, v) of vertices (nodes). We will use the term *node,* rather than *vertex*, since it is more intuitive to assign to it certain properties that are necessary at some of the steps described below. The nodes can have different flowchart shapes and have assigned connection points $(p_1, \ldots, p_n)$ where connections are attached. We will use the term *connection* for the pair (u, v) and the term *edge* for the polyline in the final drawing.

2.1. Definitions

In the presented method the flow direction F of the graph G can be either vertical (top to bottom), or horizontal (left to right), creating congruent drawings. Switching between flow directions is supported by a rotational transformation of the coordinate system. All graph drawing elements have coordinates in the coordinate system corresponding to the flow direction. When the flow direction is switched, all coordinates are recalculated using the transformation matrix. The positions of the nodes are mapped to a grid consisting of two kinds of rectangular areas: layers $I_L = (L_0, L_1, \ldots, L_p)$ across the flow direction, and lanes $I_M = (M_0, M_1, \ldots, M_q)$ along the flow direction. The nodes are positioned in the cells $C = (c_0, c_1 \ldots, c_{p+q})$ at the intersections of the layers and the lanes. Each cell position is defined by a pair (l, m) where l € L_i and m € M_j. This means that each node can be identified by the pair (l, m) of the cell in which it resides. This pair defines the *position* of the node, while its *XY* coordinates define its *location*.

The layers are separated by rectangular areas defined as layer pipes $P_L = (l_0, l_1, \ldots, l_{p+1})$, and the lanes are separated by rectangular areas defined as lane pipes $P_M = (m_0, m_1, \ldots, m_{q+1})$. Each pipe contains zero or more segment tracks $T = (t_0, t_1, \ldots, t_k)$ running lengthwise within the pipe and presumably positioned at a predetermined distance from each other. The edges are represented as polylines of segments. Each edge corresponds to a connection (u, v) between two nodes (source and target) and consists of a polygonal chain of line segments $(s_0, \ldots, s_r)$.

In the first phase, the shortest paths for routing the edges are determined by applying Dijkstra's algorithm following orthogonal graph visibility as explained later. For each edge, the segments created using the set of weighted graph vertices are assigned to layer pipes or to lane pipes correspondingly. The segments within each pipe are positioned on different segment tracks that define the segment locations within the pipe. The segment tracks are created as needed whenever a segment is assigned to a pipe. Each segment track can contain one or more non-overlapping segments. In certain cases the edges shortest paths are allowed to cross through empty cells. This approach excludes the possibility for the edges to cross graph nodes and guarantees the spacing between adjacent segments and between segments and nodes, and is the precondition for an efficient reduction of edge crossings.

The subsequent phase implements steps based on the presented heuristic algorithm to eliminate or to reduce to a reasonable minimum the number of crossings between edges. These steps include iterations for: (a) resolving the crossings between segments around the source ends of the edges, (b) resolving the crossings between segments around the target ends of the edges, and (c) resolving the crossings between middle segments in pipes and segments of adjacent edges. To eliminate these crossings, the conflicting segments are moved to adjacent segment tracks or to new parallel segment tracks created in their pipes. Since the segments are chained, moving the segments does not break the chains of segments; rather, the segments are effectively 'rubber banded' while still calculating the shortest paths of the edges. Each connection connects two nodes. Each resulting edge consists of a polygonal chain of line segments. The segments are chained in such a way that consecutive segments share their start and end points. The node at the originating end of an edge is a *source* node, and the node at the terminating end is a *target* node. Each segment chain of a routed edge starts at a connection point on the source node, referred to as an *output port*, and ends at a connection point on the target node, referred to as an *input port*. Multiple connections do not share ports on any node.

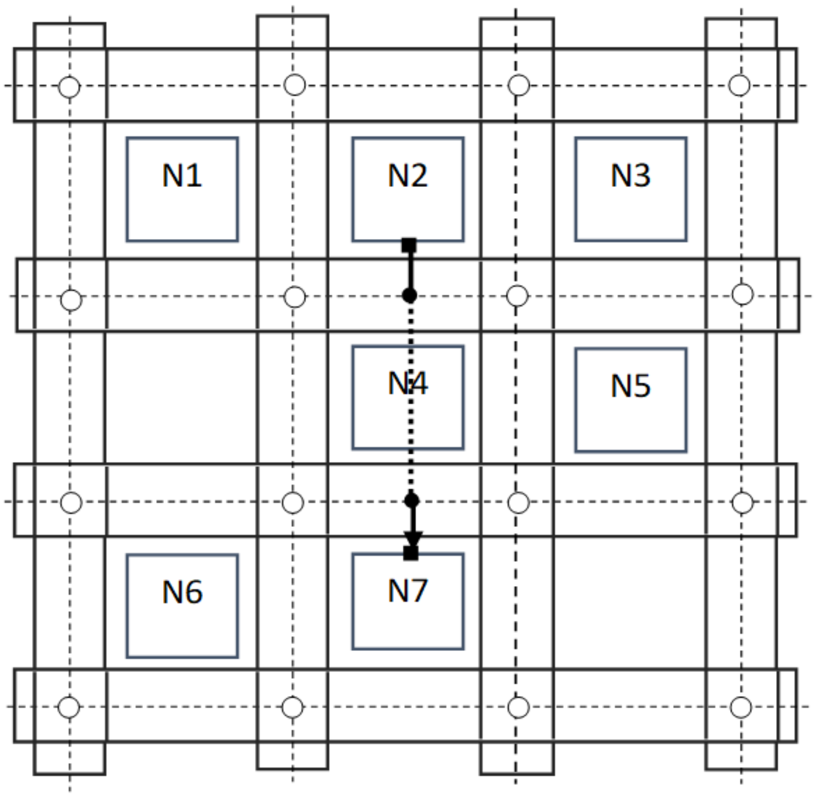

Figure 1. Rectangular grid

As illustrated in Figure 1, the first segment of an edge is referred to as a *source segment handle*, and the last segment of an edge is referred to as a *target segment handle*. The source segment handle starts at an output port on a source node and extends orthogonally to a point referred to as a *source anchor point*. The target segment handle starts at a point referred to as a *target anchor point* and extends orthogonally to an input port of a target node. The two anchor points delineate the remaining part of the segment chain of the edge. The rectangular areas around the layers are the layer pipes, and the rectangular areas around the lanes are the lane pipes. Most of the segments of the edges are laid out along the layer pipes and the lane pipes. The median line of each pipe that is running lengthwise the pipe is referred to as a *main median line*. The crossings between the main median lines of the layer pipes and the main median lines of the lane pipes are *dummy vertices* $D = (d_{00}, d_{0p}, \dots, d_{i0}, \dots, d_{ij}, \dots, d_{p+1,q+1})$, marked with circles. They are used to calculate the end points of the line segments.

The dimensions of the nodes may depend on the number of the corresponding ports, on the shape of the nodes, and possibly on some additional graphical content. The dimensions of the layers and the lanes are calculated based on the contained nodes. The dimensions of the layer pipes and the lane pipes are determined by the number of tracks. The overall size of the drawing grid is determined eventually by the dimensions of the contained drawing elements.

2.2. Routing the edges

During the edge routing stage, the set of connections E is traversed in sequence. The routing of each edge connection follows a sequence of procedural steps. The first step is to determine a new output port at the source node and a new input port at the target node. The next step is to determine a source anchor point and a target anchor point and to determine a source segment handle as the first segment of the edge and a target segment handle as its last segment. For each of the two nodes the initial locations of the anchor points are determined in two steps identical for both the source anchor point and the target anchor point: (a) finding the orthogonal projection of the respective port onto the main median line of the adjacent pipe (Figure 1), and (b) inspecting the neighbouring cells in the layer and the lane where the node is positioned. In most cases the anchor point is located on the main median line of the corresponding adjacent pipe. However, if some empty neighbouring cells are detected, the anchor point is pushed further away.

The locations of the anchor points are used to calculate the shortest path for each edge. After the edge is routed, the locations of the anchor points are adjusted and moved to the respective segment tracks where the segments are assigned. After the source segment handle and the target segment handle are determined by the locations of the anchor points, the next step is to determine the shortest orthogonal path between the two anchor points. This is achieved by applying Dijkstra's algorithm. The set of weighted graph vertices, supplied to the algorithm, includes the source anchor point as a source vertex, the target anchor point as a destination vertex, and all the dummy vertices across the grid. Running Dijkstra's algorithm produces a resulting set of vertices $W = (w_0, w_1, \dots, w_n)$, defining the shortest orthogonal path between the source vertex w_0 and the destination vertex w_n. Here w_0 is the source anchor point, and w_n is the target anchor point, and $Y = (w_1, \dots, w_{n-1}) \subset D$. The path trajectory delineates the vertices where bends are located. Each collinear group of vertices from the resulting set Y defines a new segment (w_i, w_j) along the main median line of the pipe. Each new segment is assigned to that pipe and positioned respectively on a segment track (Figure 2).

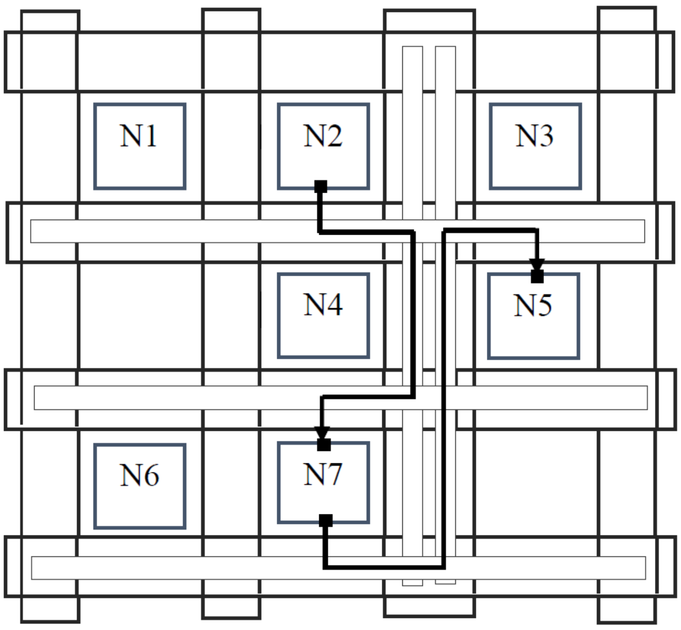

Figure 2. Routing edge connections

There are cases when there is a direct visibility between the two anchor points. In such simple cases Dijkstra's algorithm is applied to a reduced set of vertices, determined by the rectangle enclosing the source and target node.

The final step of the procedure for routing an edge is to position the created segments on the segment tracks in the pipes they are assigned to, along the calculated shortest path. At this point the locations of some anchor points may change. Initially the anchor points are positioned on the main median lines of the corresponding pipes. After the shortest path has been calculated, the locations of the anchor points are adjusted with respect to the segment tracks on which the segments are positioned. In case there is a single segment track in a pipe, the segment track coincides with the main median line of the pipe and the location of the anchor point does not change. Otherwise, the anchor point is reallocated on the respective segment track. The flowchart in Figure 3 illustrates the steps to route an edge.

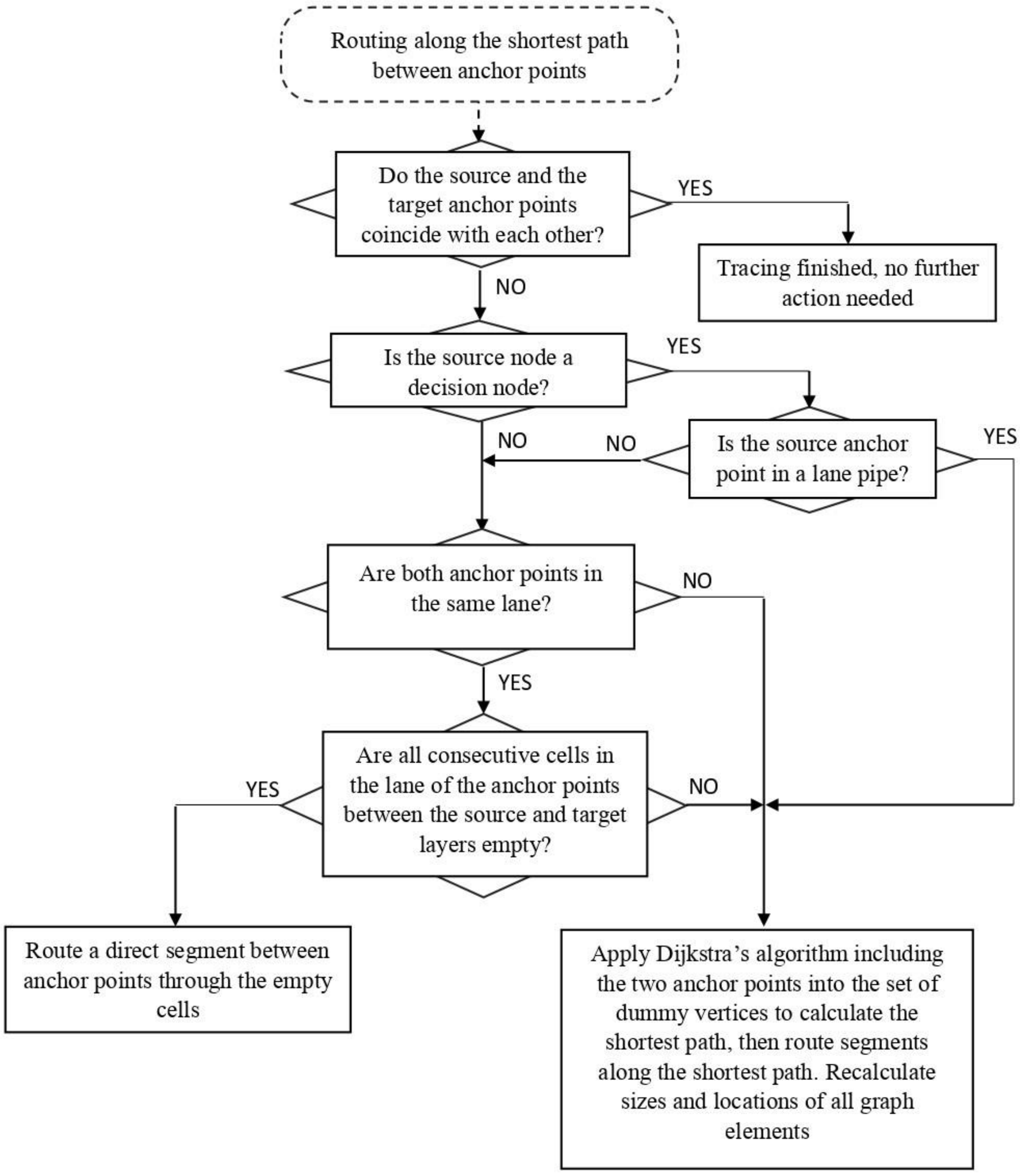
Routing along the shortest path between anchor points
Do the source and the target anchor points coincide with each other?
YES
Tracing finished, no further action needed
NO
Is the source node a decision node?
YES
Is the source anchor point in a lane pipe?
YES
NO
NO
Are both anchor points in the same lane?
NO
YES
Are all consecutive cells in the lane of the anchor points between the source and target layers empty?
YES
NO
Route a direct segment between anchor points through the empty cells
Apply Dijkstra's algorithm including the two anchor points into the set of dummy vertices to calculate the shortest path, then route segments along the shortest path. Recalculate sizes and locations of all graph elements

Figure 3. Routing edge steps

2.3. Crossing reduction

The crossings reduction starts with examination of the created set of edges *E* for crossings. The analysis is performed iteratively by steps where each step includes immediate actions to fix the detected crossings. Most of the intersections are eliminated either by swapping segment handles, or by moving some of the segments to different parallel segment tracks. The crossings reduction is performed by the following steps in this order: (a) iterative detection and fixing of the crossings between source segment handles and segments adjacent to their sibling source segment handles, (b) iterative detection and fixing of the crossings between target segment handles and segments adjacent to their sibling target segment handles, and (c) iterative detection and fixing of the crossings between some segments in pipes and segments adjacent to other segments in the same pipes. These procedures are repeated in this order until a satisfactory condition is reached.

The flowchart in Figure 4 illustrates an example of the procedure (c) for the detection and fixing of the crossings between segments in some pipes and segments adjacent to other segments in the same pipes. The procedures (a) and (b) have similar steps. The first step here is to initialize a pipe crossings counter to calculate the number of crossings. The next step is the iteration traversal of the set of edges *E* to identify a group consisting of zero or more ordered pairs of segments $S_X=((s_i, s_j), \ldots,(s_m, s_n))$ so that the segments of a pair selected from this pipe crossings group are assigned to one and the same pipe but are positioned on different segment tracks from this pipe, and at least one segment adjacent to a segment selected from the pair intersects with another segment selected from the same pair. Any adjacent segment in this case is expected to be located on a perpendicular pipe. The next step is to check the number of pairs in the pipe crossings group. If the number of pairs is equal to zero, the procedure ends by reporting zero value. If the number of pairs is greater than zero, the next step is to select the first pair from the pipe crossings group, to remove the first segment from the first segment track, to remove the second segment from the second segment track; and if the first segment does not have an overlapping conflict with any of the remaining segments positioned on the second segment track and if the second segment does not have an overlapping conflict with any of the remaining segments positioned on the first segment track, to position the first segment on the second segment track and to position the second segment on the first segment track, otherwise to add an additional segment track to the pipe that the segments from the first pair are assigned to, to position the first segment on the additional segment track, to reposition the second segment on the second segment track, to remove the first pair from the pipe crossings group and to recalculate the locations of all affected graphical elements. The next step is to check the current number of pairs in the pipe crossings group. If the current number of pairs is equal to zero, the procedure ends by reporting zero value. If the current number of pairs is greater than zero, the next step is to check if this is the first iteration of this procedure. If the result is negative (not the first iteration), the next step is to check if the number of pairs in the pipe crossings group is less than the value in the pipe crossings counter. If the result from this step is negative, the next step is to check if the number of pairs in the pipe crossings group is equal to the value in the pipe crossings counter. If the result from this step is positive, it means that the minimal number of crossings between segments in pipes and segments adjacent to other segments in the same pipes is reached, and the iterations end by reporting the number of pairs in the pipe crossings counter. This number is the actual number of unresolved crossings. If the result from the step is negative, it means that the number of pairs in the pipe crossings group is greater than the value in the pipe crossings counter and a predefined minimal number of crossings was detected in the previous iteration. The next step is to roll back the current iteration to the previous one, and to end the iterations by reporting the number of pairs in the pipe crossings counter. If the result from this step is positive (in the case this is the first iteration of the procedure), or if the result from this step is positive (meaning that the number of pairs in the pipe crossings group is less than the value in the pipe crossings counter), the next step is to set the value of the pipe crossings counter to be equal to the number of pairs in the pipe crossings group, and to return to the previous step for the next iteration.

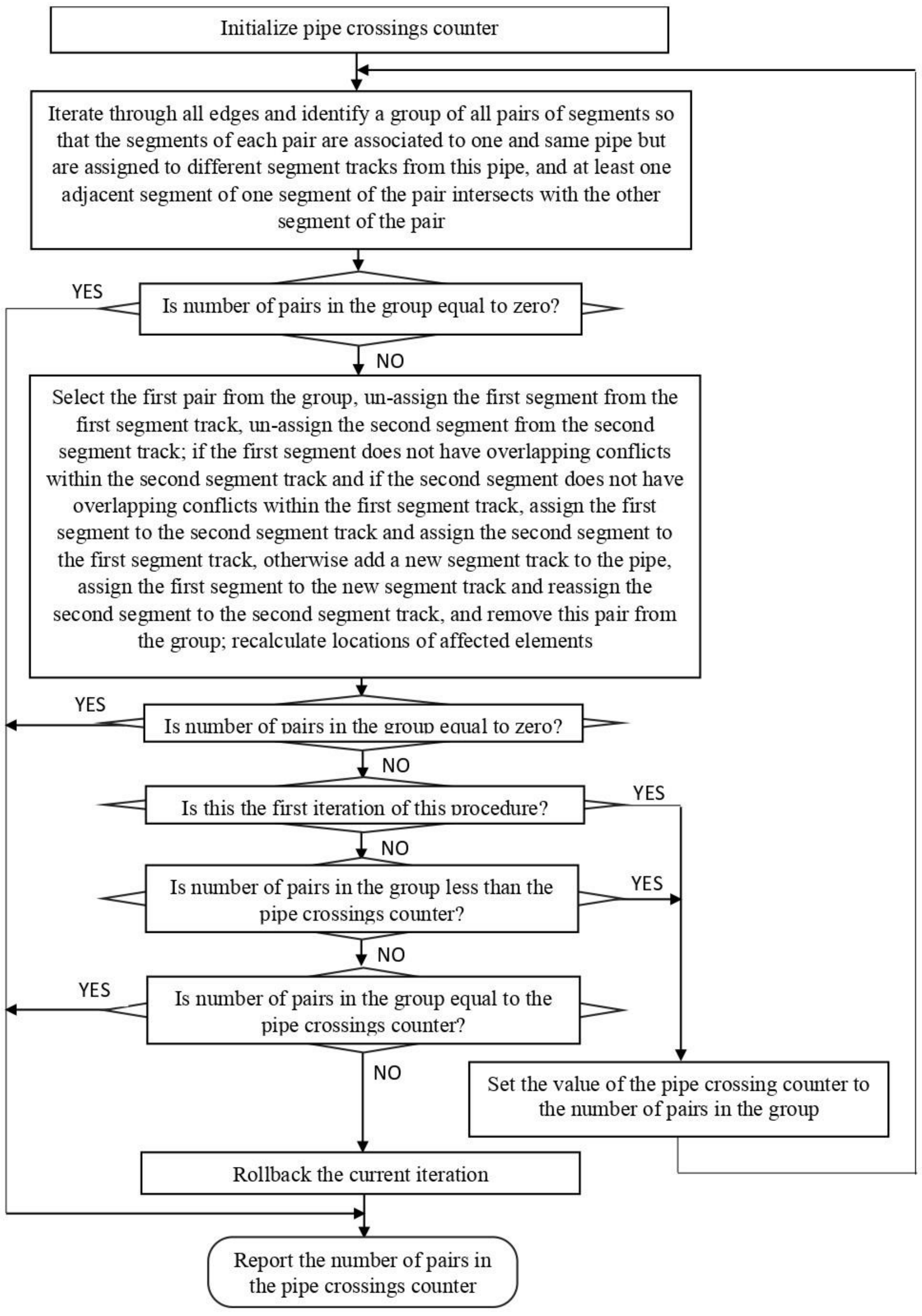
Initialize pipe crossings counter
Iterate through all edges and identify a group of all pairs of segments so that the segments of each pair are associated to one and same pipe but are assigned to different segment tracks from this pipe, and at least one adjacent segment of one segment of the pair intersects with the other segment of the pair
Is number of pairs in the group equal to zero?
YES
NO
Select the first pair from the group, un-assign the first segment from the first segment track, un-assign the second segment from the second segment track; if the first segment does not have overlapping conflicts within the second segment track and if the second segment does not have overlapping conflicts within the first segment track, assign the first segment to the second segment track and assign the second segment to the first segment track, otherwise add a new segment track to the pipe, assign the first segment to the new segment track and reassign the second segment to the second segment track, and remove this pair from the group; recalculate locations of affected elements
Is number of pairs in the group equal to zero?
YES
NO
Is this the first iteration of this procedure?
YES
NO
Is number of pairs in the group less than the pipe crossings counter?
YES
NO
Is number of pairs in the group equal to the pipe crossings counter?
YES
NO
Set the value of the pipe crossing counter to the number of pairs in the group
Rollback the current iteration
Report the number of pairs in the pipe crossings counter

Figure 4. Fixing crossings in adjacent pipes

The flowchart in Figure 5 illustrates an implementation of crossings reduction algorithm at global level with an example of an iterative sequence of all steps. The first step initializes a total crossings counter to monitor the number of unresolved edge crossings after each iteration, as well as a current crossings counter to accumulate the number of unresolved edge crossings at each step. The next step is to activate a procedure to fix crossings between source segment handles and adjacent segments. This procedure iterates through all edges to detect and fix crossings between source segment handles and segments adjacent to their sibling source segment handles. At its completion, this procedure reports the number of unresolved crossings, and this number is used to increment the value of the current crossings counter. The next step is to activate a procedure to fix crossings between target segment handles and adjacent segments. This procedure iterates through all edges in order to detect and fix crossings between target segment handles and segments adjacent to their sibling target segment handles. At its completion, this procedure reports the number of unresolved crossings, and this number is used to increment the value of the current crossings counter. The next step is to activate a procedure to fix crossings in adjacent pipes. This procedure iterates through all pipes in order to detect and fix crossings between segments in pipes and segments adjacent to other segments in the same pipes. At its completion, this procedure reports the number of unresolved crossings, and this number is used to increment the value of the current crossings counter. The next step is to check the value of the current crossings counter. If this value is equal to zero, the sequence ends by reporting the zero value. If this value is not equal to zero, the next step is to check if this is the first iteration of this sequence. If the result is negative (not the first iteration), the next step is to check if the value of the total crossings counter is less than the value of the current crossings counter. If this value is negative, meaning that the minimal total number of unresolved crossings has been reached, the iterations of the sequence end with the reporting the number of unresolved crossings. If the result from this step is positive (this is the first iteration of the sequence), or if the result from the previous step is positive (the value of the total crossings counter is less than the value of the current crossings counter), the next step is to set the value of the total crossings counter to the value of the current crossings counter, to reset the current crossings counter, and to return to the initial step for the next iteration of the sequence.

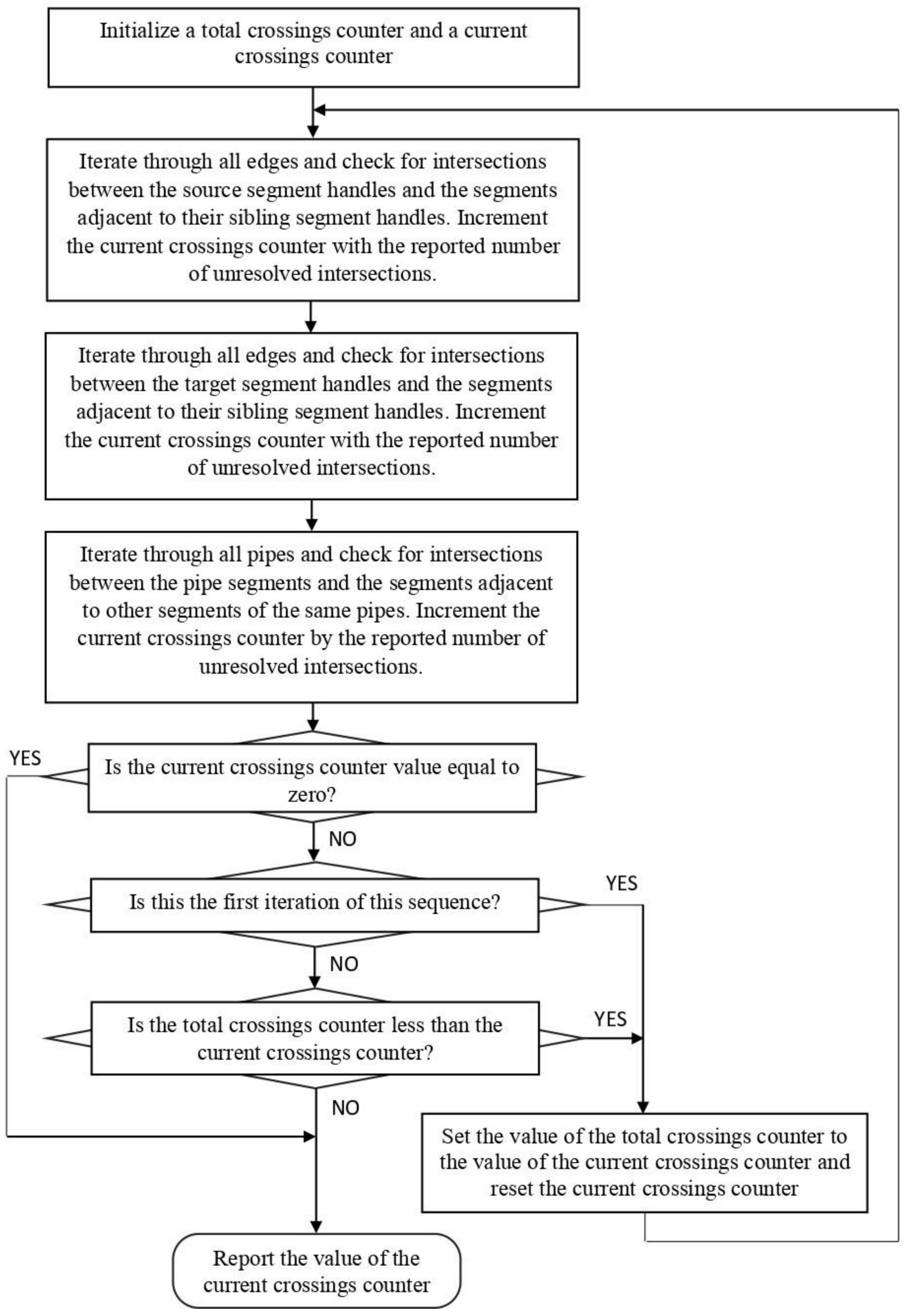
Initialize a total crossings counter and a current crossings counter
Iterate through all edges and check for intersections between the source segment handles and the segments adjacent to their sibling segment handles. Increment the current crossings counter with the reported number of unresolved intersections.
Iterate through all edges and check for intersections between the target segment handles and the segments adjacent to their sibling segment handles. Increment the current crossings counter with the reported number of unresolved intersections.
Iterate through all pipes and check for intersections between the pipe segments and the segments adjacent to other segments of the same pipes. Increment the current crossings counter by the reported number of unresolved intersections.
YES
Is the current crossings counter value equal to zero?
NO
Is this the first iteration of this sequence?
YES
NO
Is the total crossings counter less than the current crossings counter?
YES
NO
Set the value of the total crossings counter to the value of the current crossings counter and reset the current crossings counter
Report the value of the current crossings counter

Figure 5. Fixing crossings over the entire diagram

3. CONCLUSION

The described method and algorithm were implemented in JDElite Diagram Builder, a Web graph interactive flowcharting editor (www.jdelite.com). In order to build a diagram, the user only needs to drag-and-drop nodes from a palette and to enter the connections between them by mouse which is followed by automatic routing of the edges and refreshing the view. The editor is especially efficient to create medium sized complex diagrams, showing satisfactory refresh time. It shows longer refresh time for very large flowcharts.

REFERENCES

[1] C. A. Duncan and M. T. Goodrich, “Planar Orthogonal and Polyline Drawing Algorithms,” in Handbook of Graph Drawing and Visualization, R. Tamassia, Ed.; Boca Raton, FL, USA: Chapman and Hall/CRC Press, 2013, ch. 7, pp. 223-246. Available: http://cs.brown.edu/people/rtamassi/gdhandbook/chapters/orthogonal.pdf.

[2] C. D. Schulze et al., “Drawing Layered Graphs with Port Constraints,” J. Vis. Lang. Comput., vol. 25, no. 2, Apr. 2014. Available: https://www.researchgate.net/publication/258433383_Drawing_Layered_Graphs_with_Port_Constraints.

[3] D. P. Dobkin et al., “Implementing a General-Purpose Edge Router,” in Graph Drawing: 5th International Symposium, GD ‘97, Rome, Italy, September, 18-20, 1997, Proceedings, G. DiBattista, Ed.; Berlin, Heidelberg, Germany: Springer-Verlag, 1997, pp. 262-271. Available: http://dpd.cs.princeton.edu/Papers/DGKN97.pdf

[4] E. R. Gansner et al., “A Technique for Drawing Directed Graphs,” IEEE Trans. Softw. Eng., vol. 19, no. 3, pp. 214-230, Mar. 1993, doi: 10.1109/32.221135. Available: https://www.graphviz.org/Documentation/TSE93.pdf

[5] G. DiBatista et al., Graph Drawing: Algorithms for the Visualization of Graphs. Englewood Cliffs, NJ, USA: Prentice Hall, 1999. ISBN: 0-13-301615-3

[6] M. Forster, “Applying Crossing Reduction Strategies to Layered Compound Graphs,” in Graph Drawing: 10th International Symposium, GD 2002, Irvine, CA, USA, August 26-28, 2002, Revised Papers, M. T. Goodrich and S. G. Kobourov, Eds.; Berlin, Heidelberg, Germany: Springer-Verlag, 2002, pp. 276-284. Available: https://pdfs.semanticscholar.org/5d6f/de3d38417a5a92380ab09f3d84cffb8d0054.pdf

[7] M. Wybrow et al., “Orthogonal Connector Routing,” in Graph Drawing: 17th International Symposium, GD 2009, Chicago, IL, USA, September 22-25, 2009, Revised Papers, D. Eppstein and E. R. Gansner, Eds.; Berlin, Heidelberg, Germany: Springer-Verlag, 2010, pp. 219-231. Available: http://citeseerx.ist.psu.edu/viewdock/download;jsessionid=579B148C3160452F0EE0A3F0115E7A36?doi=10.1.1.159.2326&rep=rep1&type=pdf

[8] M. Cermak et al., "Edge Routing and Bundling for Graphs with Fixed Node Positions," in Proc. 15th International Conference on Information Visualisation, Jul. 2011, pp. 475-481, doi: 10.1109/IV.2011.47. Available: https://www.computer.org/csdl/proceedings/iv/2011/0868/00/06004087.pdf

AUTHOR

Jordan Raykov

Short Biography: Many years of experience in software design and engineering. Currently a software consultant at JDElite Consulting, Boulder, CO, USA