Human-Friendly Line Routing for Hierarchical Diagrams

Tobias Reinhard, Christian Seybold, Silvio Meier, Martin Glinz, Nancy Merlo-Schett
Department of Informatics, University of Zurich, Switzerland
reinhard, seybold, smeier, glinz, schett @ifi.unizh.ch

Abstract

Hierarchical diagrams are well-suited for visualizing the structure and decomposition of complex systems. However, the current tools poorly support modeling, visualization and navigation of hierarchical models. Especially the line routing algorithms are poorly suited for hierarchical models: for example, they produce lines that run across nodes or overlap with other lines.

In this paper, we present a novel algorithm for line routing in hierarchical models. In particular, our algorithm produces an esthetically appealing layout, routes in real-time, and preserves the secondary notation of the diagrams as far as possible.

1 Introduction

With the advent of UML 2.0 [9], there has been renewed interest in hierarchical models and their effective visualization. However, current modeling tool implementations rather poorly support modeling, visualization and navigation of hierarchical models (see [13] for a survey). The typical way of visualizing hierarchically structured models is by explosive zooming: when looking at the details of a particular node, the diagram with the details either replaces the previously displayed diagram or a new window is opened which supersedes the previously viewed one. In both cases, the context of the zoomed node is lost. This is bad because humans typically want to see their focus of interest in detail together with its surrounding context.

The problem of lost context in the visualization of hierarchical models can be mitigated by using fisheye zooming. Fisheye zoom algorithms show the foci of interest more detailed, whereas their context is shown with less details. In our previous work we have developed a logical fisheye visualization technique [14, 6, 2] which enables filtering, abstraction and easy navigation in hierarchical models.

However, navigating in a model by hiding or showing elements as well as generating views with fisheye zooming change the layout of the model. For the sake of model understandability, the generated layout should resemble the original layout drawn by the modeler as far as possible. This is important because a modeler builds a cognitive map [1] which is reflected in the so-called secondary notation [11]. The secondary notation consists of layout information which is reflected in the positioning, the size and other user defined and visualized properties of the model. Therefore, fully automated layout algorithms are not well-suited.

The layouting problem can be roughly divided into (a) the positioning of nodes (classes, states, activities, etc.) and (b) routing lines that connect nodes (associations, state transitions, port connections, etc.). Problem (a) has been investigated in our previous work [6, 14] where we implemented the visualization and navigation techniques described above in the ADORA tool. In this paper, we deal with problem (b): we describe a novel algorithm for routing lines in hierarchically decomposed models. Our algorithm has four distinctive properties: (i) it routes in real time whenever a modeler changes the layout of a model by navigating, creating a view or editing; (ii) it generates a graphically appealing layout (no collisions or overlaps, short paths); (iii) it preserves the secondary notation as far as possible, and (iv) it allows a modeler to modify the generated layout of a line route and preserves this modification as a new secondary notation of the connection.

The presented algorithm for line routing has been implemented in our ADORA tool [14, 13]. However, our algorithm works on any hierarchical box-and-line language, for example, UML 2.0 composite structure diagrams, activity diagrams or state machine diagrams.

The remainder of the paper is organized as follows. In section 2, we describe our line routing algorithm. Related work is discussed in Section 3. In Section 4, we summarize our results, discuss advantages and limitations and sketch our future work.

2 Line Routing for Hierarchical Elements

To maintain the readability of a diagram, it is desirable that a line between two nodes does not pass through any other nodes or overlap with other lines. As fisheye navigation requires the generation of a new diagram layout in every navigation step, automatic and fast (i.e. real-time) line routing is essential. Furthermore, when editing a diagram, the tedious task of good line routing should also be done automatically so that the modeler can concentrate on her goal of creating or modifying a model.

In contrast to other domains such as VLSI design, diagram esthetics plays an important role in the modeling domain. The lines in a diagram have to be easy to follow and add a clear meaning to the diagram. But there is only little information available about what is a "good" line [12]. In most areas that use diagrams for visualizing information, a preferred style or consensus has emerged, e.g. rectilinear lines in circuit diagrams.

We are focusing on diagrams that are incrementally created by a human user and not on diagrams that are automatically created for visualizing an existing structure. Hence, we cannot compute diagram layouts (including line routing) from scratch as done in graph drawing [4] or reengineering visualization (e.g. [5]). Instead, we must be able to adapt a given layout incrementally, preserving the secondary notation as far as possible.

In the following subsections we give a short description of our line routing algorithm. A detailed discussion of the algorithm (including a performance test) and the corresponding background work can be found in [13].

2.1 The routing problem

As an initial simplification, we consider routing of individual lines around nodes that represent obstacles; without any regard to existing lines. This is an instance of the well-known routing problem.

The routing problem has been studied extensively in VLSI design to automatically layout the wires that connect the circuit components. The first and perhaps best known routing algorithm for VLSI design is Lee's algorithm [7], which is an application of Dijkstra's breadth-first shortest path search algorithm [3] to a uniform grid. Lee's algorithm is based on the expansion of a diamond-shaped wave from the source point that continues until the target point is reached. The shortest path can be found in a second step by going back from the target point to the source point while selecting the neighbored grid cell with the lowest distance value (see [13] for a detailed description of Lee's algorithm). The algorithm always finds a solution if one exists, and ensures an optimal solution. The major drawback of this approach is that its space and runtime complexity is $O(mn)$ for a grid with $m * n$ cells.

2.1.1 Data structure

Instead of a uniform grid we use the corner stitching structure [10] as the underlying data structure for line routing. The corner stitching structure has originally been developed as an efficient storage mechanism for VLSI layout systems and has two important features: (i) All the space, whether occupied by a node or empty, is explicitly represented in the structure, and (ii) the space is divided into rectangular areas that are stitched together at their corners like a patchwork quilt.

The corner stitching structure for the four nodes in Fig. 1 is shown by the dashed lines. The space is divided into a mosaic with rectangular tiles of two types: space tiles and solid tiles. The space tiles are organized as maximal horizontal strips, i.e. no space tile ever has another space tile immediately to its right or left.

The advantage of a corner stitching structure with maximal horizontal strips over a uniform grid structure is its linear space complexity: while the number of cells in a uniform grid is determined by the size and the resolution of the grid, the maximum number of space tiles in the corner stitching structure only depends on the number of nodes. In a diagram with n nodes, there will never be more than $3n + 1$ space tiles (see [10] for a proof).

2.1.2 White space computation

Our line routing approach is divided into two completely decoupled steps: The first step computes one or multiple sequences of space tiles through which the shortest path has to pass. The second step computes the line itself, i.e. the bend points in a polyline or the curves of a spline inside the white space tiles that have been calculated in the first step.

For computing the path(s) of space tiles through which the line has to pass, we apply the fundamental wave expansion idea of Lee's algorithm to the corner stitching structure instead of a uniform grid structure.

During its expansion phase, our algorithm computes a distance value for the space tiles of the structure. Due to the non-uniform tile size of the corner stitching structure, it is not possible to use the distances from the source point to the tiles as distance values, because there may be multiple different values for one tile. We therefore use a combination of the source distance and the distance to the target point which are both measured in the Manhattan distance[1]. Furthermore, for each tile the algorithm computes the point P inside the tile where the distances are actually measured. For the actual search, an ordered data structure (e.g. heap, priority queue) denoted as Ω is used. Below, we present an informal description of the algorithm:

1. Construct the corner stitching structure and determine the tile T_{start} that contains the source point s and the tile T_{end} that contains the target point t.

2. Set the point P for T_{start} to the source point s, the source distance of T_{start} to 0 and the distance of T_{start}

[1]The Manhattan distance, also known as the L_1-distance, between two points P and Q is defined as the sum of the lengths of the projections of the line segments onto the coordinate axes: $\lambda(P, Q) = |x_P - x_Q| + |y_P - y_Q|$

to the Manhattan distance between P and the target point t. Insert T_{start} into Ω.

3. As long as Ω contains tiles: Remove the tile T with the lowest distance value from Ω. If this tile is the tile T_{end}, a path has been found. Otherwise, calculate for each neighboring space tile T_{next} the point P_{next}, i.e the point inside T_{next} closest to the point P of the current tile T. Compute for each of these neighboring tiles two distance values: The *source distance* σ is calculated by adding the Manhattan distance between the point P and the point P_{next} to the source distance of T. The *distance* δ is calculated by adding the Manhattan distance between P_{next} and the target point t to σ. Equations 1 and 2 show the calculation of the source distance σ and the distance δ for the tile T_{next} relative to the tile T, whereas $\lambda(P_1, P_2)$ denotes the Manhattan distance between the points P_1 and P_2:

$$\sigma(T_{next}) = \sigma(T) + \lambda(P, P_{next}) \tag{1}$$

$$\delta(T_{next}) = \sigma(T_{next}) + \lambda(P_{next}, t) \tag{2}$$

   Check whether the calculated distance value for T_{next} is lower than a previously calculated value for T_{next}. If so, update the distance δ and source distance σ values of T_{next}. Insert the tile T_{next} into Ω.

4. The sequence(s) of space tiles that constitute the shortest path can be found, in analogy to Lee's and Dijkstra's algorithm, by moving back from T_{end} to T_{start} by repeatedly selecting a neighbored tile that has the same distance value δ. If multiple neighbors have the same value, there exist multiple solutions and the current tile is a branch.

Fig. 1 shows the corner stitching structure after the second iteration through step 3 of the algorithm. The current tile T corresponds to T_6 and the distance values σ and δ for the tiles T_4, T_5 and T_7 are calculated. According to equation 1, the source distance σ for T_7 is equal to the sum of the source distance of T_6 (which is zero in this case) and the Manhattan distance between the points P and P_{next} in T and T_{next}, respectively, which is 36. The distance δ is calculated by adding the Manhattan distance between the point P_{next} and the target point t, which is 576, to the source distance σ.

2.1.3 Line routing

The algorithm for finding the space tiles that the shortest path has to pass through is completely decoupled from the algorithm that does the actual routing (i.e. computes the coordinates for drawing the line). Hence, different algorithms that produce different styles of lines can be implemented on top of the algorithm described in Section 2.1.2. Fig. 2 shows three different line routing styles: a rectilinear polyline (which we have currently implemented), an unconstrained polyline and a spline.

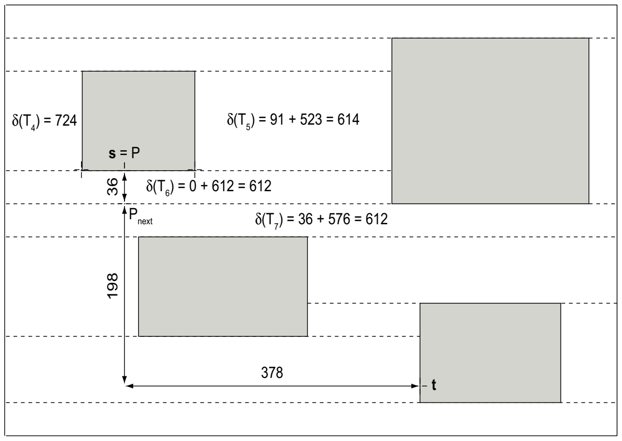

Figure 1. Routing algorithm

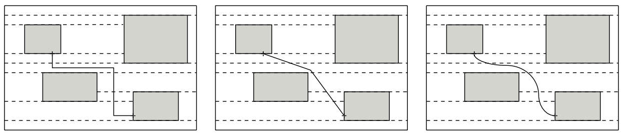

Figure 2. Different line routing styles

2.2 Line crossings

By extending the corner stitching structure and the algorithm of Section 2.1.2, we can avoid line crossings during the routing of a line. We extend the corner stitching structure by a third tile type: the line tile. Line tiles are space tiles weighted by a constant *cost factor* α. The algorithm now calculates cost values instead of distance values for the tiles. A higher cost factor α of a tile increases the costs for a line to pass through this tile. We are therefore transforming the source distance σ and the distance δ into a *source cost* ω and a *cost* γ. Equations 3 and 4 show the extension of equations 1 and 2 with the cost factor α:

$$\omega(T_{next}) = \omega(T) + \alpha * \lambda(P, P_{next}) \tag{3}$$

$$\gamma(T_{next}) = \omega(T_{next}) + \lambda(P_{next}, t) \tag{4}$$

2.3 Preserving Secondary Notation

In connection with line routing, the secondary notation has two aspects: The routing algorithm has to tolerate some user influence, i.e. the line should not be routed completely automatically. And once defined, the secondary notation of a line has to be preserved in case of layout changes.

The algorithm that has been described so far lets the user select where the source and target points are anchored on the source and target node. Furthermore, by varying the cost factor α, the user can define to what extent the algorithm should avoid line intersections. If there exist multiple shortest paths, it is also possible to let the user select a path manually or add a selection function to the algorithm.

But the definition of the secondary notation by the user does not necessarily end with the application of the routing algorithm. The user may change the line that has been proposed by the routing algorithm by moving the segments of a rectilinear line or the bending points of a unconstrained polyline. These changes can be preserved when the line has to be rerouted in a later layout modification step by using the geometric information (e.g closest neighbor) stored in the corner stitching structure (see [13] for the details).

3 Related Work

The existing approaches to visualizing and editing hierarchical models are very limited concerning node positioning and employ rather primitive line routing techniques only [13]. Significant work on line routing, however, has been done in domains other than modeling where line routing is a major concern. In the field of automatic *graph drawing* the routing of the edges that connect the nodes is an integral part of the node placement algorithm (e.g. [4]). There is no need for incremental layout adaptation or preservation of a layout produced by a human. The wire routing problem that occurs in*VLSI design* has to deal with a fixed placement of the circuit components and therefore is similar to our line routing problem. The most important routing algorithms in this domain are those developed by Lee [7] and Soukup [15] and their variants. Due to their runtime complexity, wire routing algorithms cannot be computed in real time on a personal computer. The geometric shortest path problem in *computational geometry* has many applications in robotics, geographic information systems and diagram drawing. The best known approach constructs a visibility graph [8] and computes the shortest path on this graph according to Dijkstra's approach [3]. The avoidance of line crossings can't be integrated directly into the visibility graph and therefore has to be done in a second step after the routing.

4 Conclusions

We have presented a novel line routing algorithm which is based on the idea of using Lee's algorithm on a corner stitching data structure. To achieve our goals of appealing layout and preservation of secondary notation, we have adapted and extended the basic algorithm. All presented algorithms have been implemented in our ADORA modeling tool written in Java, which is able to generate views from hierarchical models. We have applied our algorithm to example models with encouraging results, both concerning usability and routing speed.

A performance test [13] has demonstrated that our algorithm is fast enough to route a large number of lines in real time (up to 150 lines per diagram) even though we have implemented the algorithm without special optimizations.

Currently, our line routing algorithm is limited to rectilinear routing. Splines may be an alternative. However, determining intersections is more complex for splines. Label positioning is also not implemented yet. In our ongoing research, we want to investigate to what extent nicer esthetics justify a more complex routing algorithm.

References

[1] D. V. Beard and J. Q. Walker II. Navigational Techniques to Improve the Display of Large 2-D Spaces. *Behaviour & Information Technology*, 9(6):451–466, 1990.

[2] S. Berner, S. Joos, M. Glinz, and M. Arnold. A Visualization Concept for Hierarchical Object Models. In *Proceedings of the Thirteenth IEEE Conference on Automated Software Engineering (ASE '98)*, pages 225–228, 1998.

[3] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerical Mathematics*, 1:269–271, 1959.

[4] P. Eades. A Heuristic for Graph Drawing. *Congressus Numerantium*, 42:149–160, 1984.

[5] M. Eiglsperger, M. Kaufmann, and M. Siebenhaller. A Topology-Shape-Metrics Approach for the Automatic Layout of UML Class Diagrams. In *SoftVis '03: Proceedings of the 2003 ACM Symposium on Software Visualization*, pages 189–198, 2003.

[6] M. Glinz, S. Berner, and S. Joos. Object-Oriented Modeling with ADORA. *Information Systems*, 27(6):425–444, 2002.

[7] C. Y. Lee. An Algorithm for Path Connections and its Applications. *IRE Transactions on Electronic Computers*, EC-10(3):346–365, September 1961.

[8] N. J. Nilsson. A Mobile Automaton: An Application of Artificial Intelligence Techniques. In *Proceedings of the First International Joint Conference on Artificial Intelligence*, pages 509–520, 1969.

[9] OMG. *Unified Modeling Language: Superstructure Version 2.0*. Document formal/05-07-04, Object Management Group, 2005.

[10] J. K. Ousterhout. Corner Stitching: A Data-Structuring Technique for VLSI Layout Tools. *IEEE Transactions on Computer-Aided Design CAD*, 3(1):87–100, January 1984.

[11] M. Petre. Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming. *Communications of the ACM*, 38(6):33–44, 1995.

[12] H. C. Purchase. Which Aesthetic Has the Greatest Effect on Human Understanding? In *Proceedings of the 5th International Symposium on Graph Drawing*, pages 248–261, 1997.

[13] T. Reinhard, C. Seybold, S. Meier, M. Glinz, and N. Merlo-Schett. A Novel Algorithm for Line Routing in Hierarchical Diagrams. Technical Report ifi-2006.08, University of Zurich, 2006.

[14] C. Seybold, M. Glinz, S. Meier, and N. Merlo-Schett. An Effective Layout Adaptation Technique for a Graphical Modeling Tool. In *Proceedings of the 25th International Conference on Software Engineering*, pages 826–827, 2003.

[15] J. Soukup. Fast Maze Router. In *Proceedings of the 15th Conference on Design Automation*, pages 100–102, 1978.