Interactive, Orthogonal Hyperedge Routing in Schematic Diagrams Assisted by Layout Automatisms

Stefan Helmke(✉), Bernhard Goetze, Robert Scheffler, and Gregor Wrobel

R&D Department of Graph-Based Engineering Systems, Society for the Advancement of Applied Computer Science, Volmerstraße 3, 12489 Berlin, Germany
{Helmke,goetze,scheffler,wrobel}@gfai.de

Abstract. Schematic diagrams are used in graph-based engineering systems. They focus mainly on the structure of the design object. Graph-based engineering systems help to solve a concrete design task. This is primarily realized by the application of domain-specific languages. The layout of schematic diagrams is of particular importance, and a neat representation is desirable. But automatically generated layouts cannot always fully match the intention of a modeler. To improve automatic layouts and enable a user-specific representation, an algorithm that allows interactive changes of the orthogonal hyperedge geometry was implemented. In this paper, we present this algorithm and give an overview of such interactions. Additionally, several reductions of the hyperedge geometry are shown. Furthermore, a local, automatic routing considering interactions on the hyperedge geometry is presented. The consideration of domain-specific semantics and the possibility of interactive changes is a new approach. All algorithms were implemented in a self-developed software framework.

Keywords: Graph drawing · Orthogonal hyperedge routing · Interactive changes

1 Introduction

Schematic diagrams are used in graph-based engineering systems (GES). They focus mainly on the structure of the design object. GES help to solve a specific design task. This is primarily realized by the application of domain-specific languages. A GES is model-centered and uses a meta-model. The design model contains real or abstract objects. It is graphically represented in the GES and allows interactive editing.

The interactive treatment of these models, i.e., the design process, is a major concern of our research. We developed procedures that support the modeler in the design process, and we implemented these procedures in a framework for GES [1]. Graphical languages, i.e., schematic diagrams, and their layout are of particular importance. Principles for designing effective visual notation [2] for the depiction of vertices and edges are used to reach a neat representation. Furthermore, common methods of graph drawing like the placement of vertices and the routing of edges are applied. These representations need to be embedded in their specific problem domain to acquire good readability, i.e., semantic

A. Basu et al. (Eds.): Diagrams 2021, LNAI 12909, pp. 20–27, 2021.
https://doi.org/10.1007/978-3-030-86062-2_2

transparency. This greatly improves the clarity, recognizability, and interpretability of complex models.

In the past, automatic routing algorithms were developed [3] to fulfill domain-specific requirements. But automatically generated layouts cannot always fully match the modeler's intention. Therefore, automatic layouts, e.g., activated by the movement, mirroring, or rotating of vertices, should consider interactive changes made by the modeler on the hyperedge geometry, while simultaneously preserving a neat layout.

The consideration of domain-specific semantics and the possibility of interactive changes is a novel approach. In this paper, we focus on an orthogonal hyperedge representation to ensure good readability, and we describe automatic routings that consider manual changes.

In the following sections, we refer to the internal data structure ELADO (Extended Layout Data Model) [1] to describe hypergraphs. In this data structure, vertices and hyperedges are not immediately connected, they are connected via so-called pins. The hyperedge geometry contains branch points and segments, while segments contain bends and (horizontal or vertical) segment parts. To avoid redundant information, segments contain no transit points. Transit points are pseudo-bends connecting two adjacent segment parts with the same orientation: either horizontal or vertical. Additionally, hyperedges are connected and acyclic, i.e., the hyperedge geometry is a tree. The vertices of this tree are bends, branch points, and pins; the edges of this tree are segment parts.

Section 3 describes and classifies possible, interactive changes of hyperedges by moving horizontal and vertical segment parts. This results in reductions of unwanted states. In Sect. 4, we explain how automatic routings executed after the movement of vertices consider the interactive changes mentioned in Sect. 3. Therefore we improved the geometrically stable routing PartRoute [4], which is a modification of the automatic routing OrthoRoute presented in [3]. Section 5 concludes the paper.

2 Related Work

Fundamental algorithms for drawing graphs were shown in [5], which is widely considered as a standard reference for graph drawing. A common set of aesthetic criteria was defined to improve the readability of diagrams: the minimization of crossings, bends, the length of the edges, or the area occupied by a drawing [6]. In [7] metrics were proposed to quantify the aesthetic quality of diagrams. In [8] the authors show an algorithm minimizing the number of hyperedge crossings. Additionally, in [9] the authors show an approach that dynamically reorders the vertices within the layers to further reduce the number of crossings. In [10] two crossing counting algorithms that predict the number of crossings between orthogonally routed hyperedges are presented. But besides that, there are only a few procedures for orthogonal hyperedge routing. An example is [11], where two automatic routings are given, and one semi-automatic routing, which improves the layout by local transformations, is added. A routing for directed hypergraphs with layers is shown by Sander [12].

3 Interactive Routing of Hyperedges

We now observe interactive changes of the hyperedge geometry. It is allowed only to move horizontal segment parts vertically and vertical segment parts horizontally. There are interactions that change the geometry of a hyperedge but not its structure. We call them trivial interactions (Fig. 1 (a)). These change only the length of segment parts, i.e., segment parts can contract or expand. There are also nontrivial interactions that are characterized by the transformation, genesis, or removal of bends, branch points, or segment parts (Fig. 1 (b)). Therefore, it is necessary to classify all the possible interactions.

First of all, we denote by *sp* the moved segment part, or more precisely the segment part that is about to be moved. Furthermore, we define the satellites s_1 and s_2 of *sp*. These are the vertices incident to *sp*. Satellites can be bends, branch points, or pins. We call the line segment between a satellite before an interaction and the same satellite after an interaction the satellite's trajectory.

We now study two kinds of interactions: interactions not causing collisions and interactions causing collisions with adjacent segment parts (Fig. 1 (a), (b)). These interactions depend completely on the satellites and the segment parts that are adjacent to *sp*. The type of a satellite is then given by the triple $(n_1, n_2, n_3) \in \{0, 1\}^3$ where n_1 is the number of adjacent segment parts in the direction of movement of *sp*, n_2 is the number of adjacent segment parts in the opposite direction of movement, and n_3 is the number of adjacent segment parts parallel to *sp*. It follows that there are 7 types of satellites. According to ELADO, the type (0, 0, 1) (transit point) is not permitted.

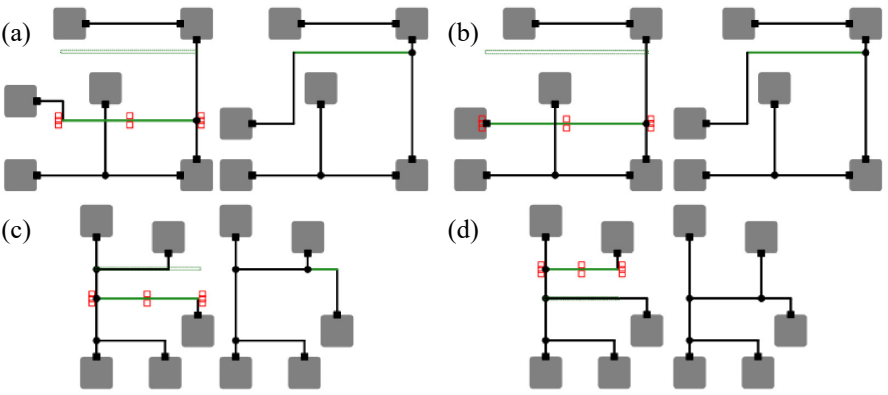

Fig. 1. (a) The right satellite collides with an adjacent segment part (trivial interaction). (b) The left satellite is a pin. Two bend points and a segment part are generated (nontrivial interaction). The right satellite collides with an adjacent segment part. (c) The left satellite collides with a branch point, *sp* collides with a bend. (d) The left satellite collides with a branch point. The right satellite collides with a segment part.

A segment part *sp* can also be moved so that *sp* or the satellites collide with other bends, branch points, or segment parts that are not adjacent to satellites (Fig. 1 (c), (d)). The following types of collisions are then allowed: a satellite collides with a bend or a branch point, a satellite collides with a segment part, or *sp* itself collides with a bend or a branch point.

First, we classify the bends or branch points colliding with a satellite depending on the segment parts that are incident to those bends or branch points. We define the type of such a colliding bend or branch point as the quadruplet $(n_1, n_2, n_3, n_4) \in \{0, 1\}^4$ where n_1 is the number of incident segment parts in the direction of movement of *sp*, n_2 is the number of incident segment parts in the opposite direction of movement, n_3 is the number of incident segment parts colliding with *sp*, and n_4 is the number of incident segment parts that are parallel to a potentially colliding segment part. There are 9 possible types of colliding bends or branch points. According to ELADO, the types (0, 0, 0, 0) (isolated point); (1, 0, 0, 0), (0, 1, 0, 0), (0, 0, 1, 0), (0, 0, 0, 1) (pins); and (1, 1, 0, 0), (0, 0, 1, 1) (transit points) are not permitted.

Second, we describe the bends or branch points colliding with *sp* itself depending on the segment parts that are incident to them. The type of such a colliding bend or branch point is then given by the triple $(n_1, n_2, n_3) \in \{0, 1\}^2 \times \{0, 1, 2\}$ where n_1 is the number of incident segment parts in the direction of movement of *sp*, n_2 is the number of incident segment parts in the opposite direction of movement, and n_3 is the number of incident segment parts colliding with *sp*. There are 6 possible types. According to ELADO, the types (0, 0, 0) (isolated point); (1, 0, 0), (0, 1, 0), (0, 0, 1) (pins); and (1, 1, 0), (0, 0, 2) (transit points) are not permitted. If a bend or branch point collides with *sp*, we call it an explicit collision. A bend or branch point that is located on a satellite's trajectory can also collide with a segment part. We then call it an implicit collision.

Third, we class the segment parts colliding with a satellite depending on their position relative to *sp*. It is easy to see that there are only three cases. A colliding segment part and *sp* can be parallel or orthogonal. In a special case, a satellite can also be moved exactly onto a crossing.

Without a grid, it is not easy for users to force collisions, because it is nearly impossible to move a segment part exactly onto an object. In this case, the position of *sp* is automatically corrected if the distance between *sp* and a colliding object is smaller than a defined parameter ε. It is possible that multiple collisions appear simultaneously. Then the collision with the smallest distance to the original position of *sp* is chosen to correct *sp*.

In the first phase of the algorithm, the observation phase, the types of the satellites are determined. According to the aforementioned classification, the colliding objects and the bends and branch points on the trajectories of the satellites are determined. In the second phase, a canonical form is created. The first aim is to reduce the types of collisions so that satellites can only collide with bends, branch points, or temporary transit points. If a satellite collides with a segment part (Fig. 1 (d)), then a temporary transit point is generated on this segment part, and this segment part is divided into two. If a satellite is moved onto a crossing, then a branch point is generated on this crossing, and the two involved segment parts are divided into four. The position of these generated transit points and branch points equals that of the satellite. The second aim of this phase is to connect *sp* with bends or temporary transit points. If a satellite has the type (0, 0, 0), i.e., the satellite is a pin (Fig. 1 (b)), a transit point is generated on *sp* depending on the exact position of the mouse click and the length of *sp*. This transit point divides *sp* into two segment parts. If the type is not (1, 0, 0) or (0, 1, 0), i.e., the satellite is not a bend, a transit point is generated on the satellite including a segment

part with length 0 to maintain the orthogonal layout. In the next phase, the satellites are moved in the direction of movement, and the segment parts originally generated with length 0 are expanded. The aim of this phase is to consider and handle collisions. If a bend or branch point *b* is located on a satellite's trajectory (i.e., implicit collision), and there is a segment part in the opposite direction of movement, then *b* is transformed into a branch point. In addition, the colliding segment part is divided into two segment parts, and these are connected with *b*. If a satellite collides with a bend, a branch point (Fig. 1 (c), (d)), or a temporary transit point *b* (Fig. 1 (a), (b), (d)), then *b* is connected with *sp*. If the satellite is connected with a segment part in the opposite direction of movement, then this segment part is connected with *b*, too. If *sp* collides with a bend or branch point *b* (Fig. 1 (c)), then *b* is transformed to a branch point. In addition, the colliding segment part is divided into two segment parts, and these are connected with *b*. At the end of this phase, the unwanted structures in the hyperedge geometry are removed: Transit points, geometrically identical segment parts (to avoid interfering lines), and segment parts with length 0 are removed; branch points with valence 2 are transformed into bends.

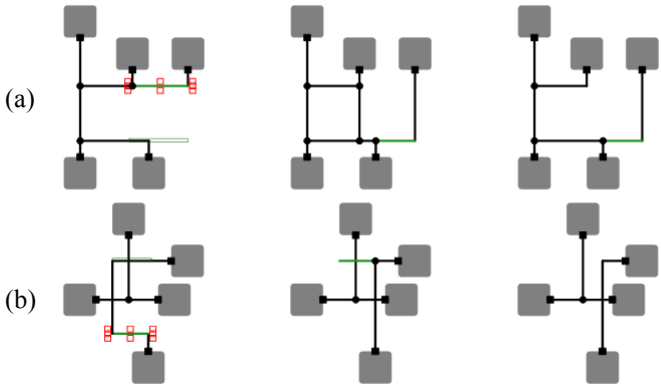

Fig. 2. (a) Automatic reduction of a cycle. (b) Automatic reduction of a stump.

In rare cases, some interactions can cause collisions that result in further unwanted states in the hyperedge geometry: cycles and stumps[1], which have to be removed automatically (Fig. 2). Of course, it is not predetermined how to reduce a cycle. The algorithm prefers to remove new, implicitly generated connections.

4 Automatic Routing Considering Manual Changes

In Sect. 3, we observed interactive changes of the hyperedge geometry to improve automatic routings. If another local, automatic routing is executed, e.g., by moving a vertex, users expect it to consider interactive changes. Otherwise, all interactive changes are lost. We improved the PartRoute method [4] to solve this task and explain the algorithm in the following.

[1] A stump is defined as a leaf, which is not a pin.

First, the PartRoute method is improved by choosing a flexible docking object; see Fig. 3. We define the critical segments as those segments that are incident to the pins of the moved vertex v, and we call the pins of v critical pins. The algorithm removes the critical segments and resulting transit points, while the rest of the hyperedge geometry remains. To find a suitable docking object, we search for the bend or branch point with the minimal Euclidean distance to the critical pin, and we denote it by b. Then a routing is executed in every cardinal direction. The type of the routing depends on the free directions of b. In the free directions, b is a potential docking object. But if a segment part is incident, i.e., this is not a free direction, a movable branch point is added on this segment part. This branch point is another potential docking object. Horizontal segment parts require two routings starting in this branch point with a northern or a southern segment part, and vertical segment parts require two routings starting in this branch point with an eastern or a western segment part. Therefore, there are up to 8 routings and 4 potential docking objects if b has degree 4 and no free directions. From these routings, the best one is chosen depending on the number of bends, the length of the critical segment, and the distance to other vertices. The segment part with the minimal Euclidean distance to the critical pin may also have a smaller distance to the critical pin than the nearest bend or branch point b, though this segment part is not incident to b. In this case, this segment part is an additional, potential docking object.

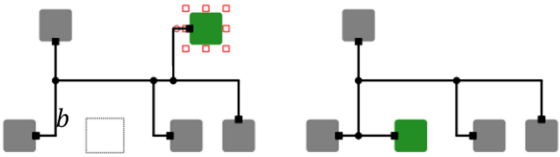

Fig. 3. PartRoute with flexible docking objects. The nearest bend or branch point is b. It has two free directions (east and south), in each of which a routing is performed. Additionally, two routings are executed on the western and northern segment part each time. The best routing is the one starting in b with an eastern segment part.

Having improved the PartRoute method by choosing flexible docking objects, the next task was to improve it in such a way that interactive changes of the hyperedge geometry are considered when a moved segment part sp is located on a critical segment. W.l.o.g., sp is vertical. If sp is horizontal the procedure is equivalent. The x-coordinate of sp should be fixed, while the length of sp is still flexible (Fig. 4 (a), (b)). The locked state of sp is dissolved when the routing is unsatisfying in terms of the length and the number of bends (Fig. 4 (c)). The specific approach depends on the position of the critical pin relative to sp.

We denote by s_1 the satellite with a smaller distance (in the tree) to the critical pin. The other satellite is denoted by s_2. Because sp is vertical the x-coordinate of s_1 is fixed, but the y-coordinate of s_1 is flexible. The coordinates of s_2 are also fixed. After a vertex v is moved, the critical pin is connected with s_1 in a way that this path minimizes the number of bends and the length of the path. This determines the y-coordinate of s_1. If the order of the y-coordinates of s_1 and s_2 does not change, then sp contracts, expands, or

does not change its length (Fig. 4 (a), (b)). Otherwise, the locked state of *sp* is dissolved, and the improved PartRoute method is used (Fig. 4 (c)).

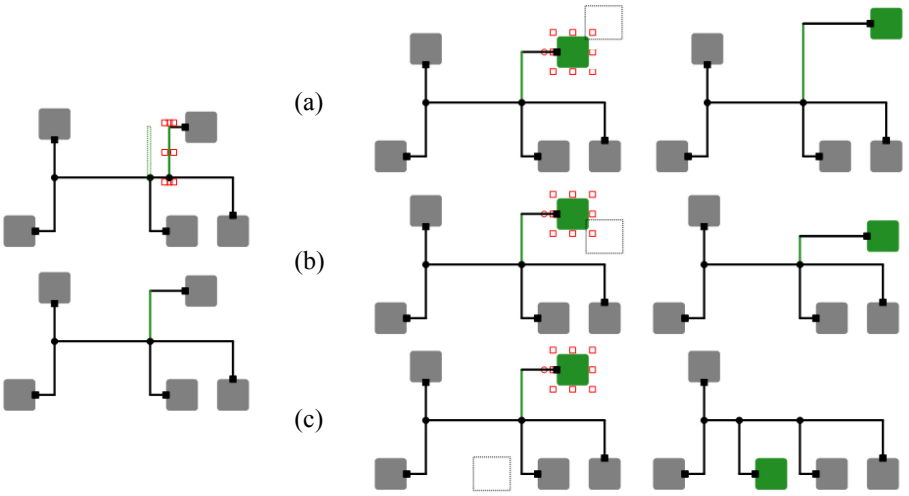

Fig. 4. PartRoute considering moved segment part *sp*. (a) *sp* expands. (b) *sp* contracts. (c) The locked state of *sp* is dissolved. The y-coordinate of s_1 would be smaller than the y-coordinate of s_2, while the order before the movement of v was reversed.

5 Conclusion

In Sect. 3, we show several possibilities of how users can manipulate an automatic routing by interactively changing the hyperedge geometry. A result of these manual changes is the automatic reduction of a hyperedge geometry to avoid unwanted states. In Sect. 4, we give several improvements to the PartRoute method for a more aesthetic routing minimizing the length and the number of bends. Furthermore, we present a modification of the PartRoute algorithm that considers manual interactions.

One challenge is that the PartRoute method and the OrthoRoute method differ significantly in the degree of geometric flexibility. Therefore, we want to improve the PartRoute algorithm so that it can automatically determine how geometrically flexible a routing can be depending on the specific situation. A future task is to create a geometrically flexible, automatic routing considering interactive changes—a dynamic, partial routing.

Acknowledgments. This R&D project of the Society for the Advancement of Applied Computer Science (GFaI) is supported by the funding program Innovation Competence (INNO-KOM) of the German Federal Ministry for Economic Affairs and Energy (BMWi), based on a resolution of the German Parliament.

References

1. Wrobel, G., Ebert, R.-E., Pleßow, M.: Graph-based engineering systems – a family of software applications and their underlying framework. Electronic Communications EASST Volume 6: Graph Transformation and Visual Modeling Techniques 2007, vol. 6 (2007). https://doi.org/10.14279/tuj.eceasst.6.50
2. Moody, D.: The "physics" of notations: toward a scientific basis for constructing visual notations in software engineering. IIEEE Trans. Software Eng. **35**, 756–779 (2009). https://doi.org/10.1109/TSE.2009.67
3. Goetze, B.: OrthoRoute – Ein gitterloses Verfahren zur Generierung orthogonaler Verbindungen in Schaltplänen. In: Roller, D., Opletal, S. (eds.) Tagungsband Workshop Elektrotechnik CAD 2007, pp. 13–32. Shaker, Aachen (2007)
4. Scheffler, R., Goetze, B.: PartRoute – Geometrisch stabiles Routing orthogonaler Verbindungen in Schaltplänen. In: Roller, D., Opletal, S. (eds.) Tagungsband Workshop Elektrotechnik CAD 2007, pp. 33–42. Shaker, Aachen (2007)
5. Di Battista, G., Eades, P., Tamassia, R., Tollis, I.G.: Graph Drawing. Algorithms for the Visualization of Graphs. Prentice Hall, Upper Saddle River (1999)
6. Tamassia, R., Di Battista, G., Batini, C.: Automatic graph drawing and readability of diagrams. IEEE Trans. Syst. Man Cybern. **18**, 61–79 (1988). https://doi.org/10.1109/21.87055
7. Purchase, H.C.: Metrics for graph drawing aesthetics. J. Vis. Lang. Comput. **13**, 501–516 (2002). https://doi.org/10.1006/jvlc.2002.0232
8. Eschbach, T., Günther, W., Becker, B.: Orthogonal hypergraph routing for improved visibility. In: Proceedings of the 14th ACM Great Lakes Symposium on VLSI 2004, Boston, MA, USA, 26–28 April 2004, pp. 385–388 (2004). https://doi.org/10.1145/988952.989045
9. Eschbach, T., Guenther, W., Becker, B.: Orthogonal hypergraph drawing for improved visibility. JGAA **10**, 141–157 (2006). https://doi.org/10.7155/jgaa.00122
10. Spönemann, M., Schulze, C.D., Rüegg, U., von Hanxleden, R.: Counting crossings for layered hypergraphs. In: Dwyer, T., Purchase, H., Delaney, A. (eds.) Diagrams 2014. LNCS (LNAI), vol. 8578, pp. 9–15. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44043-8_2
11. Wybrow, M., Marriott, K., Stuckey, P.J.: Orthogonal hyperedge routing. In: Cox, P., Plimmer, B., Rodgers, P. (eds.) Diagrams 2012. LNCS (LNAI), vol. 7352, pp. 51–64. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31223-6_10
12. Sander, G.: Layout of directed hypergraphs with orthogonal hyperedges. In: Liotta, G. (ed.) GD 2003. LNCS, vol. 2912, pp. 381–386. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24595-7_35