Shortest Path Approach to Edge Routing

Jiri Dokulil, Jana Katreniakova, David Bednarek
University of Vienna, Vienna, Austria
Comenius University, Bratislava, Slovakia
Charles University, Prague, Czech Republic
jiri.dokulil@univie.ac.at, katreniakova@dcs.fmph.uniba.sk, bednarek@ksi.mff.cuni.cz

Abstract

Traditionally, drawing of edges is performed together with drawing of nodes. However, there are situations where positions of the nodes are fixed, e.g., when the positions are defined by the user or a separate algorithm. An example of this situation is a database schema editor, where user positions the nodes (i.e., visual representations of definitions of individual database tables) according to their meaning, for example grouping them according to subdomains of the problem. In this case, we only need to draw the edges but we must do that in such a way that the lines that represent these edges do not cross the rectangles that represent the nodes – we need to perform some kind of edge routing. This paper describes an algorithm that performs edge routing in such a way that the lengths of the polylines it produces are minimal. We also describe several ways of improving the performance of the basic algorithm so that it can be used even for interactive graph visualization and manipulation, which is necessary in our scenario. Then, we show several post-processing steps that are used to turn the results of the algorithm into a usable visualization.

1 Introduction

There is a huge number of different applications of graph drawing. They impose different limitations on inputs and outputs of the graph drawing algorithms so there is no definitive solution that would fit every situation. One, less common and less explored, set of problems are those where we already know the drawing of the nodes of the graph. The shape and position of the visual elements that represent the nodes in the drawing have already been supplied by an outside source – either a user or software.

A number of such use cases come from the rapidly expanding area of parallel and distributed computing, where branched workflows, pipelines, execution plans and other parallel computation schemes are often inspected or designed by humans and computer-assisted drawing forms an invaluable tool in the design of parallel applications especially in the area of e-science. From the other use cases, we selected the widely understood visualization of database schemas as a running example.

In all the cases mentioned above, the nodes of the graph (i.e. the tables in a database schema or various units of computation in parallel computing) have some naturally preferred layouts for the human operator, like clustering the tables together according to a sub-problem that they are used for. In addition, to maintain human understanding, the layout shall not change much when nodes or edges are added or removed. On the other hand, the users often don't want to bother with positioning of the edges (i.e. the foreign keys in a database schema) as they expect that the layout of the edges be inferred from the layout of the nodes.

The edges in the graph are drawn as (poly)lines connecting the related nodes. Quite often, the only criterion for the routing of these lines is that it should be "nice". This makes the task ideal for graph drawing techniques.

In the following text, we present one approach to the problem. We took an inspiration from an already existing solution for visualization of interconnected class instances which uses an algorithm that routes the polylines in such a way that they have minimal possible length [6]. The advantage of the approach is predictability and stability (the same input produces identical results, similar inputs usually produce similar results), which is useful when the target audience are software developers and the software serves as a tool where aesthetics are not important. The disadvantage of that algorithm is the fact that it can only be used on small graphs due to its time complexity. There are improved versions of this approach, but none of them completely fit our needs. So, we have come up with several ways to significantly speed up the original algorithm (in the average case, not asymptotically). Even though the worst case performance is still poor, we believe (and our experiments confirm it) that it works well in practical applications.

However, before the results can actually be presented to the user, there are several post processing steps that should be performed to improve readability and aesthetics of the drawing. We describe two of these steps, one of which is necessary (and sufficient) and the other is optional.

We also provide a way to efficiently handle the situation where the position of one node is being changed by the user (update handling). This is a common scenario: the user has just decided to move one of the tables in the schema designer to a position that suits him or her better. It turns out that it is not necessary to redraw the whole graph.

First, we will discuss related work. Then, in Section 3, we describe the original algorithm and the improvements that we have done. Section 4 deals with implementation issues, especially with ways of speeding up the evaluation. Section 5 describes the post processing steps. Then, we briefly discuss the implementation and measured performance. The last section concludes the paper and provides some ideas for future work.

2 Related work

Graph drawing algorithms mostly solve the problem of drawing the whole graph – positioning of nodes is interconnected with drawing of edges and one algorithm handles both subproblems. This is not possible in our case. We cannot change the positions of edges even if we could create "nicer" drawing of edges in the new layout. Therefore we can only find routes for drawing of edges in the free space between the drawn nodes. Moreover, we do not have any further information about the node positions, except the basic assumption that the rectangles do not overlap.

There are several basic options of solving this kind of problems: path-finding algorithms, force-directed algorithms, heuristics based on rerouting of edges, and shortest-path algorithms.

2.1 Path-finding algorithms

Algorithms based on robot motion planning [9, 5] try to find an effective path among the obstacles in the plane. The polygon which forms the free space in the plane is first triangulated (or otherwise subdivided). Then, the algorithm uses centers of the triangles to find a suitable path for the robot. It is the shortest path in the dual graph (tree) of the triangulation [9]. This can also be used for edge-routing. However, it does not produce the shortest possible path.

Other algorithms (e.g., [14] used for example in [7]) use the path through a maze which is constructed on the free space. They are much faster, but produce only orthogonal drawing of edges.

2.2 Force-directed algorithms

The force-directed algorithms mostly use springs or other forces between nodes and edges to achieve the best positioning of both nodes and edges. However, these forces can also be used to remove edge-node or even edge-edge intersections [2, 8]. These solutions use some other routing algorithm as the first step and then use the force-directed approach to get more aesthetic drawing.

There are many variants of these algorithms, for example constrained force-directed approach [3] that tries to preserve pre-defined node positions to some degree.

2.3 Rerouting of edges

The edge-routing algorithm [1] was designed to handle a general drawing of any graph where nodes are drawn as rectangles (only constrained by the requirement that there is at least $\delta > 0$ free space between the nodes). It finds the solution by incrementally rerouting the polyline to avoid edge-node crossings. It is a heuristic and thus does not provide any guarantees for the optimality of edge-drawing.

There are more ways of heuristically rerouting edges [10], but we have chosen this algorithm for its simplicity and implemented it as a part of our effort to improve performance of our solution (for more details, see sections 3.3 and **??**).

2.4 Shortest-path algorithms

Routing edges along the shortest path has already been tried. In [6], the author only needed to visualize small graphs, so the implemented solution did not scale well with the size of the graph. The biggest performance issue is the construction and maintenance of the visibility graph. In [13], the authors describe a way to significantly speed up this process by very efficiently managing changes to the visibility graph. We took a different path by trying to eliminate the need to build the whole visibility graph, rather than improving the performance of the building process. But some of their ideas are close to the implementation improvements described in Section 4, especially 4.2.

Unfortunately, the drawings produced by that algorithm cannot be used for our use-case, since it draws edges in such a way that they overlap, making it impossible to always distinguish the individual edges. But, it might be possible to combine our approach with the performance improvements that they use.

There is also a different way of approaching the whole problem. It is possible to slightly relax the requirement of finding the shortest path and only require a good approximation. In [4], the authors present one such solution. They are able to speed up the drawing by an order of magnitude, while keeping the mean increase of the length of the polylines at around 20%. But it always redraws the whole graph, even if there is just a partial update.

3 Algorithms

The following sections describe the algorithms from the theoretical point of view. We describe the terminology, the problem and the various algorithms that may be used to solve the problem.

3.1 Terminology

In the following text, *graph* always refers to a non-directed, finite graph $G = (V, E)$. The drawing of the graph $\Gamma(G)$ is a function over nodes and edges, that maps each node of the graph to a rectangle positioned in 2D space with its sides parallel to either x or y axis. The function Γ is naturally extended to sets of nodes and edges. We assume that Γ is defined in such a way that the rectangles that correspond to nodes do not overlap. In fact, we assume that there is $\delta > 0$ free space between the rectangles.

Further symbols and functions used in the text are:

- $\overline{x\, y}$ denotes the line connecting points x and y,
- $corners(n)$ (n is a node) is the set of (four) points that lie at the corners of $\Gamma(n)$,
- $center(n)$ is the center of $\Gamma(n)$,
- $|p|$ is the length of the polyline p.

3.2 The problem

In our case, we want to find an optimal (shortest) polyline p defined by points $(p_1, \ldots, p_k)$ that connects two nodes n_1 and n_2. In the following text, we assume that we connect the centers of the nodes, but any point that falls within the rectangle $\Gamma(n)$ would work just the same.

We say that the polyline p is *valid for* R, where R is a set of non-overlapping rectangles that includes $\Gamma(n_1)$ and $\Gamma(n_2)$ if the following conditions hold:

- no line $\overline{p_i\, p_{i+1}}$ crosses any rectangle from R for $2 \leq i \leq k-2$,
- $\overline{p_1\, p_2}$ only crosses $\Gamma(n_1)$,
- $\overline{p_{k-1}\, p_k}$ only crosses $\Gamma(n_2)$,
- no bend (i.e., $p_2, \ldots, p_{k-1}$) lies in $\Gamma(n_1)$ or $\Gamma(n_2)$.

In other words, the polyline does not cross any rectangle except that the first segment crosses $\Gamma(n_1)$ and the last segment crosses $\Gamma(n_2)$.

A polyline p valid for R is *optimal* iff there is no shorter polyline valid for R that connects the same two points. Note that we omit "for R" if a statement holds for any applicable set – a set of non-overlapping rectangles from $\Gamma(V)$ that includes $\Gamma(n_1)$ and $\Gamma(n_2)$.

Lemma 1 For any valid polyline p, there is a valid polyline $p' = (p'_1, \ldots, p'_{k'})$ such that:

- $p_1 = p'_1$ and $p_k = p'_{k'}$,
- $|p'| \leq |p|$,
- $\forall x \in (p'_2, \ldots, p'_{k'-1}) \exists n : x \in corner(n)$.

In other words, for any valid polyline, we can find a polyline, that connects the same points, is the same length or shorter, and only bends at the corners of rectangles. A physical interpretation is that if we replaced the rectangles with solid blocks of material and the polyline with a tensed strip of rubber, then the rubber strip would contract so that it only bends at the corners of the solid blocks.

As a result of this, we can limit our search for the optimal polyline to polylines that only bend at corners of rectangles. This fact was used to create an algorithm for edge routing [6], that works by creating a visibility graph of all corners and then finding the shortest path through this graph using Dijkstra's algorithm – the edge weight is naturally defined as the distance between the connected points (corners). This algorithm will be called SC (Shortest path using Corners) in the rest of the text.

The *corner visibility graph* $CVG(\Gamma(G), n_1, n_2)$ for the drawing $\Gamma(G)$ of the graph $G = (V, E)$ and for nodes n_1, n_2 from G is the graph (CVG_V, CVG_E) where $CVG_V = \bigcup_{n \in V} corner(n) \cup \{center(n_1), center(n_2)\}$ and CVG_E is the union of the following sets:

- $\{\{v_1, v_2\} : v_1 \in CVG_V \wedge v_2 \in CVG_V \wedge v_1 \neq v_2 \wedge \overline{v_1\, v_2}$ does not cross any rectangle from $\Gamma(G)\}$,
- $\{\{v_1, v_2\} : v_1 = center(n_1) \wedge v_2 \in CVG_V \wedge v_1 \neq v_2 \wedge \overline{v_1\, v_2}$ does not cross any rectangle from $\Gamma(G) \setminus \{\Gamma(n_1)\}\}$,
- $\{\{v_1, v_2\} : v_1 \in CVG_V \wedge v_2 = center(n_2) \wedge v_1 \neq v_2 \wedge \overline{v_1\, v_2}$ does not cross any rectangle from $\Gamma(G) \setminus \{\Gamma(n_2)\}\}$,
- $\{\{v_1, v_2\} : v_1 = center(n_1) \wedge v_2 = center(n_2) \wedge v_1 \neq v_2 \wedge \overline{v_1\, v_2}$ does not cross any rectangle from $\Gamma(G) \setminus \{\Gamma(n_1), \Gamma(n_2)\}\}$.

The optimal valid polyline is then defined by the shortest path from $center(n_1)$ to $center(n_2)$ in $CVG(\Gamma(G), n_1, n_2)$. The disadvantage of this approach is the time complexity of the algorithm which also results in poor performance of real-world applications even on small graphs.

3.3 Problem reduction

Two main sources of poor performance of the original algorithm are the following: first, building the visibility graph is very time-consuming, since we have to test visibility (the fact, whether a line crosses any rectangle) for each pair of corners. Second, running Dijkstra's algorithm on such a large graph also takes considerable time.

We can significantly reduce the problem if we have some upper bound of the length of the optimal polyline p. An example of such estimation is the situation, where we have a valid polyline p' that connects the same two endpoints. Obviously, the optimal valid polyline cannot be any longer, since it is defined as the shortest possible valid polyline (with the two defined endpoints).

It is clear that the optimal polyline cannot contain any point x such that $|\overline{p_1\, x}| + |\overline{x\, p_k}| > |p'|$ since that would make it longer than p' which is a contradiction.

The algorithm for edge routing can be modified by using any other edge routing algorithm $route(\Gamma(G), n_1, n_2)$ that produces valid polylines like this:

1. $limit := |route(\Gamma(G), n_1, n_2)|$,
2. define CVG' as the induced subgraph of $CVG(\Gamma(G), n_1, n_2)$ where $CVG'_V = \{c \in CVG_V : |\overline{center(n_1)\,c}| + |\overline{c\,center(n_2)}| \leq limit\}$,
3. the optimal polyline is defined by the minimal path from $center(n_1)$ to $center(n_2)$ in CVG'.

3.4 Lazy algorithm

Running the Dijkstra's algorithm on a reduced visibility graph helps with one of the sources of poor performance. But the other one, which is the time required to build the visibility graph still remains. This can be improved by a major modification of the algorithm. The basic principle is the same as the SC algorithm, but the visibility graph is built in a lazy manner – we only build the part of the graph that is needed at the moment. For this reason, we call the algorithm LSC (Lazy SC).

In the following, $path(CVG')$ denotes the polyline that was defined by the shortest path from $center(n_1)$ to $center(n_2)$ in CVG'.

```
1. $V := \{n_1, n_2\}$
2. $CVG' := CVG(\Gamma(V), n_1, n_2)$
3. $p := path(CVG')$
4. $\Gamma(n)$ := one of the rectangles from $\Gamma(G)$ crossed by p
5. if no $\Gamma(n)$ exists, $p$ is the result (end algorithm)
6. add $n$ to $V$
7. goto 2
```

The second step may be modified in the same way that was used in Section 3.3 to only include corners that are close enough. In step 4, we choose any single rectangle from a set of possible candidates. There is no simple optimal strategy for selecting the "best" rectangle – it is easy to find counter-examples for all options like the closest rectangle to origin, the rectangle in the middle, etc. For this reason, we decided to use the first candidate that we can find and end the search at that moment.

The LSC algorithm also finds an optimal valid polyline (see Appendix D).

4 Implementation

The task of creating a working implementation of all the algorithms is straightforward. However, since our goal is to create an interface that would quickly respond to user's actions, we have to build it in a more sophisticated manner.

4.1 Indexing

One of the common operations is to check whether a line crosses any rectangle that represents a node (crossing look-up). For example, this is used to build the visibility graph. A simple solution would be to take each rectangle and check, whether the line crosses it or not. But to build the full corner visibility graph, this operation is performed for (nearly) each pair of points, where a point is either a corner or center of a node. It may not be necessary to check all pairs that include a node center – we only need to include those, where the center is actually used as a endpoint of a polyline. In other words, there is an edge adjacent to the node represented by the rectangle. But the worst case may be very likely – for example, in database schema design tool, it is quite likely that each table is involved in at least one parent-child relationship (foreign key). On the whole, we are likely to test close to $(5n)(5n-1)/2$ lines to check whether they cross a rectangle that represents a node. This would necessitate $n(5n)(5n-1)/2$ tests of line-rectangle crossing.

The number of crossing look-ups can be reduced by the LSC algorithm, although the worst-case scenario remains the same. The cost of each look-up can be reduced by a spatial index. However, traditional indexes were not designed for queries in the form of a line [11].

One solution is to use a straightforward grid-based index [12], where the space is evenly covered by a square grid. Each square contains an information about all rectangles that overlap the square. The crossing look-up is performed by finding all squares that are crossed by the line and then creating a union of all rectangles associated to these squares. Then each of these candidates is tested to check whether it is really crossed by the line or not.

Obviously, in the worst case situation, we still have to perform the same number of tests as we would do without the index. But in real applications, the index can significantly improve the overall performance.

4.2 Updates

In some relevant scenarios, we may be faced with a situation where the position of one node is modified and we need to update the drawing of the graph. We could recalculate the optimal polyline for each edge, but that may not always be necessary.

There are three classes of edges that we have to update, when a node n is moved to a new location:

1. edges incident to n,
2. edges that are represented by a polyline that crosses the new location of n,
3. edges that had their drawing affected by the node n (before n was moved).

It is easy to identify edges that belong to the first two classes. The third class is slightly more difficult. It contains all edges such that the LSC algorithm used the node n (i.e., n was added to the V set used in the algorithm). We can store the set V for each edge and use it to find edges that can be updated (this is called *invisibility graph* in [13]).

Note that it is not sufficient to just update edges that directly bend at one of the corners of $\Gamma(n)$. An example of such situation is the scenario, where $\Gamma(n)$ obstructed a possible route for the polyline in such a way, that it forced the algorithm to take a completely different route, away from $\Gamma(n)$.

Also note that the update process requires the program (option 2 in the list) to find all edges that are represented by a polyline which passes through a certain area. This can either be done by brute force or by an index – the problem is very similar to the one described in Section 4.1 and can be solved by similar means.

4.3 Information sharing

When the LSC algorithm is executed for an edge, it builds a graph that is closely related to the maximal corner visibility graph $CVG(\Gamma(G), n_1, n_2)$. However, it is not a subgraph of the full graph. This is due to the fact that the "local" graph only computes the visibility with regard to the current contents of the $\Gamma(V)$ set. So it may declare that two corners can see each other, even though the line that connects them is obstructed by a rectangle from $\Gamma(G) \setminus \Gamma(V)$.

If we modify the representation of the corner visibility graph by adding the list of all rectangles that cross each link, we can overcome this problem and reuse the information that was computed for one edge in the computation of the following edges.

4.4 Parallel computation

In the basic form of the LSC algorithm, the routing of one edge is completely independent on routing of any other edge, so they can be executed in parallel to further reduce the time that the user has to wait before a drawing is generated.

However, if information sharing described in the previous section is used, special care has to be taken to synchronize generation of the global visibility graph. This can be achieved by proper use of locks and if the locks are sufficiently fine-grained (e.g., for each corner or node, rather than one global lock) it will not limit scalability by much, since collisions would be rare thanks to the fact that the number of nodes will typically be much larger than the degree of parallelism.

4.5 Overlapping nodes

In the theoretical part of the text, we always assumed that the rectangles that represent the nodes do not overlap. But in reality, to handle user's input, when the user changes the position of the nodes, we need to handle even such situations – we cannot force the user to move nodes in such a way that they never overlap throughout the whole action. The actual implementation can draw edges in any situation, but if some rectangles overlap, the edges adjacent to nodes represented by these rectangles may be drawn in a way that does not meet our criteria for edge – the polyline may not be optimal or even valid (it may cross some of the rectangles). The simplest way of achieving this behavior is to tune the Dijkstra's algorithm in such a way that it finds a path through the visibility graph even if there shouldn't be one. We add the obstructed links to the graph but with such high cost that they can only be used if there is no other way.

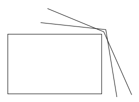
Figure 1: Example of bad corner sorting

5 Post-processing

The algorithm described in Section 3 does not produce results that could be used directly to visualize a database schema. The improvements described in the previous section only affect the performance and (in one case) "pathological" cases. In this section, we present two post processing steps that are applied to the result of the routing algorithm in order to make edges distinguishable (corner sorting) and improve the visual appearance (corner smoothing). The first step is necessary, otherwise the user will often be unable to find out, which pairs of nodes are connected and which are not.

5.1 Corner sorting

If two (or more) edges bend at the same corner, both of these bends lie at exactly the same coordinates, which in turn means that four segments (two for each edge) terminate at those coordinates. The user would not be able to identify correctly pair the segments.

So, for each corner, we take all polylines that bend at that corner plus all polylines that pass near the corner (within a pre-defined distance) without bending there. We add a new bend to the latter polylines – we make them bend at the corner they were passing.

Then, we take all of the n bends and move them to slightly different locations near the corner of the node. To do so, we need to do two things: (a) create a set P of n different positions and (b) assign bends to that positions. The problem is, that this process may introduce new edge-edge crossings (see Figure 1). We try to do the step (b) in such a way that it minimizes the problem.

The set P is created by selecting evenly spaced points on the line segment that starts and the corner a ends at a pre-defined distance from the corner in the direction “away from the corner” (along the $(1, 1)$ vector for the lower-right corner in traditional screen coordinates, i.e., with x-axis pointing right, y-axis pointing to the bottom; other corners are analogous).

To assign bends to positions, we first need to define ordering on the bends. Originally, we wanted to solve it by transforming the problem of minimizing the number of edge-edge crossings to finding a minimal permutation. But it turned out, that a much simpler and faster solution is capable of providing decent results. First, we split the plane to two sub-planes along the “away from the corner” axis. Each polyline has one segment in one plane (“left”) and another in the other (“right”). We count the number of distinct directions of all “left” segments and the same number for “right” segments. Whichever side has more distinct directions is the winner. Then, we order the segments on the winning side according to the angle they form with the “away from the corner” axis.

Finally, we assign positions from P to the bends in this order. The first bend (smallest angle) gets the point that is the farthest from the corner. In case of a draw, the order is arbitrary, but it should be stable (i.e., it should not change between multiple executions of the algorithm).

5.2 Corner smoothing

In order to make the drawing more appealing, the bends on the polylines are transformed into Bezier curves. We only transform a small region close to the actual bend, so that the overall characteristics of the drawing are maintained.

The curve is defined by a starting point $P1$, which is on the line in δ distance before the corner (internally, we maintain an implicit orientation of the edge and the polyline), end point $P4$, which is on the line in δ distance after the corner, and two control points $P2$ and P that are both placed at the original bend.

6 Implemenation and example

We have implemented all of the described algorithms and improvements. On the whole, the best performance was achieved in a variant that used all improvements except for the edge index. The cost of maintaining the index is not justified by reduced cost of edge lookup. The resulting performance is sufficient to draw graphs with tens of nodes and hundreds of edges in real time. For graphs with hundreds of nodes, it takes several seconds to compute the whole drawing, but the updates are still handled in fractions of seconds.

An example output from the algorithms is shown in Figure 2. It is only a mock-up of a databse design tool, but the graph drawing algorithms are fully implemented.

7 Conclusions and future work

We have shown several ways in which the basic idea of routing edges by finding the shortest path in the visibility graph can be made to run much faster. As a result, the algorithms could be used to implement an interactive application, which should be able to respond to user’s actions in real time. The main improvement comes from the “lazy” algorithm, which eliminates the need to build the whole visibility graph. We have also described several ways in which the performance can be further improved by efficient implementation of the algorithm and post-processing steps that transform the drawing into to make it more useful.

There are two main directions that we would like to explore in the future. First, it may be possible to most of the techniques of speeding up visibility graph construction [13] into our algorithms. Second, we would like to further improve the design of the schema editor, starting with the problem of displaying edge labels (names of the relations, possibly even more detailed information).

Acknowledgment

This work was supported in part by the grant P103/13/08195S of the Grant Agency of the Czech Republic.

References

[1] Jiri Dokulil and Jana Katreniakova. Edge routing with fixed node positions. In *IV ’08: Proceedings of the 2008 12th International Conference Information Visualisation*, pages 626–631, Washington, DC, USA, 2008. IEEE Computer Society.

[2] Tim Dwyer, Kim Marriott, and Michael Wybrow. Integrating edge routing into force-directed layout. In *IN: PROC. 14TH INTL. SYMP. GRAPH DRAWING (GD 06). VOLUME 4372 OF LECTURE*. Springer, 2007.

[3] Tim Dwyer, Kim Marriott, and Michael Wybrow. Topology preserving constrained graph layout. In Ioannis G. Tollis and Maurizio Patrignani, editors, *Graph Drawing*, volume 5417 of *Lecture Notes in Computer Science*, pages 230–241. Springer, 2008.

[4] Tim Dwyer and Lev Nachmanson. Fast edge-routing for large graphs. In David Eppstein and Emden R. Gansner, editors, *Graph Drawing*, volume 5849 of *Lecture Notes in Computer Science*, pages 147–158. Springer, 2009.

[5] Sanjiv Kapoor and S. N. Maheshwari. Efficient algorithms for euclidean shortest path and visibility problems with polygonal obstacles. In *Symposium on Computational Geometry’88*, pages 172–182, 1988.

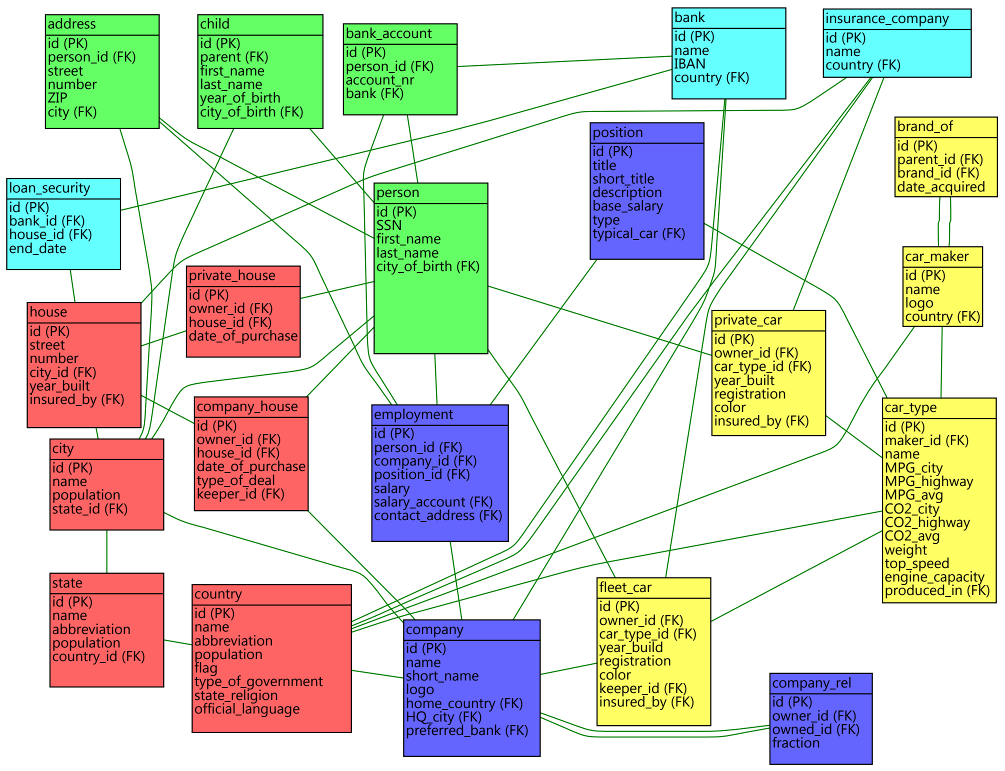

Figure 2: Example of a database schema with edges drawn using our algorithm and both post-process steps

[6] Martin Konicek. *Debugger Frontend for the SharpDevelop IDE*. Master thesis, Charles University in Prague, 2011.

[7] Pushpa Kumar, Kang Zhang, and Mao Lin Huang. From tree to graph - experiments with e-spring algorithm. In Mao Lin Huang, Quang Vinh Nguyen, and Kang Zhang, editors, *Visual Information Communication*, pages 41–63. Springer US, 2010.

[8] Wei Lai and Peter Eades. Removing edge-node intersections in drawings of graphs. *Inf. Process. Lett.*, 81:105–110, January 2002.

[9] D. T. Lee and F. P. Preparata. Euclidean shortest paths in the presence of rectilinear barriers. *Networks*, 14(3):393–410, 1984.

[10] K. Miriyala, S.W. Hornick, and R. Tamassia. An incremental approach to aesthetic graph layout. In *Computer-Aided Software Engineering, 1993. CASE '93., Proceeding of the Sixth International Workshop on*, pages 297–308, 1993.

[11] Philippe Rigaux, Michel Scholl, and Agnes Voisard. *Spatial Databases: With Application to GIS*. Morgan Kaufmann, 2001.

[12] Hanan Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1989.

[13] Michael Wybrow, Kim Marriott, and Peter J. Stuckey. Incremental connector routing. In *IN: PROC. 13TH INT. SYMP. ON GRAPH DRAWING (GD05). VOLUME 3843 OF LNCS*, pages 446–457. Springer, 2006.

[14] yFiles. Class library. http://www.yworks.com/en/products_yfiles_about.html.