Navigation in RDF data

Jiří Dokulil, Jana Katreniaková
Faculty of Mathematics and Physics,Charles University, Prague, Czech Republic
Faculty of Mathematics, Physics and Informatics, Comenius University, Bratislava, Slovakia
jiri.dokulil@mff.cuni.cz, katreniakova@dsc.fmph.uniba.sk

Abstract

There are already several tools available that are capable of visualizing RDF data. The problem with RDF data is that they tend to be very large. To handle the data, the visualizers cannot display the whole data but rather need to use some kind of navigation. This paper describes our approach to the navigation, which was designed specifically with the preservation of the user's mental map in mind. We also compare our approach to the other visualizers.

1 Introduction

The Semantic web [1] idea is already well established, as well as some of the standards that accompany it. One of those standards is the RDF [2] data format which is intended to be the low-level format for semantic data. By their very nature, the RDF data for an oriented, labeled graph. This may be used to fight one problem – the RDF data often tend to be large, complex and hard to read and explore when serialized to some text-based format (especially in the case of RDF-XML). If we present the data visually, we may be able to give the user much better idea about what is in the data. But to handle data that may contain millions or more of nodes and edges, the visualization itself is not sufficient and some kind of navigation is necessary.

The Section 2 gives a brief overview of our approach to drawing subgraphs of the whole data. The key part of this paper is the Section 3 which explains the way we let the user navigate the data. Section 4 explores other approaches to navigation in RDF data.

2 Visualization of RDF data

To present the RDF visually, we can not use visualization alone. The graph is too big, containing millions of nodes and edges. Although there are visualizations handling data that big, their purpose is to only somehow suggest the overall structure of the data – an example are visualizations of web page relations or large social networks. But we need a detailed display showing individual nodes and their connections.

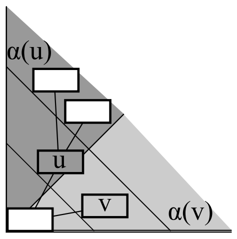
For this reason, we use navigation in our RDF visualizer. But for any navigation, we need visualization first. But in turn, we make use of the navigation for the visualization. Because the user navigates the data by adding connected nodes to the already visible part of the graph, it creates a rooted tree (that we call a *navigation tree*). And our visualization is based on drawing of that tree.

One aspect, important for both visualization and navigation, is the way we draw the nodes. They are drawn as rectangles. The content of the rectangle is label of the node and list (possibly incomplete if they are too numerous) of incoming and outgoing edges. This is called *node merging* and affects the visualization by forcing the nodes to have variable height (the number of edges varies) and width (the length of the edge labels varies).

We use layered drawing to draw the navigation tree. This means that all nodes that have the same distance from the root are displayed on the same layer. The layers in our case are lines connecting $(0, r(L))$ and $(r(L), 0)$ and nodes are placed with their lower left corner on the line. The value $r(L)$ is called *radius* of the layer L.

The visualization algorithm always starts by placing the root of the navigation tree to the coordinate origin. Then all children of the root are positioned on the first layer. The radius of the layer is made big enough for all the children to fit onto the layer without overlapping each other and also big enough to be beyond the rectangle representing the root node. The children are placed evenly along the layer.

The process is nearly the same for further layers, except there is so called *angle of influence* α. Each node in the tree is assigned an angle of influence and it is an angle where all descendants (not only children) must fit.



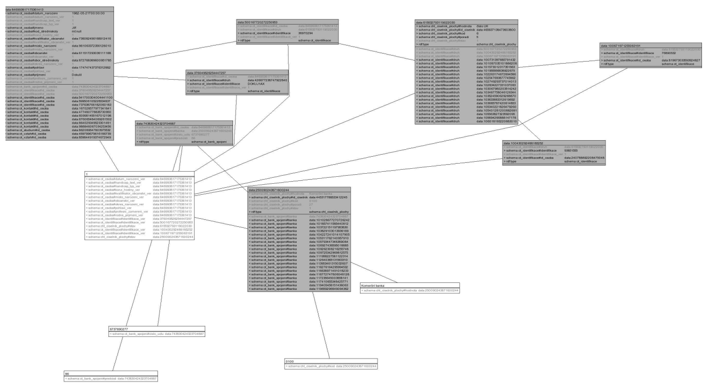


Figure 1: Example of triangle layout

Do note that the node itself need not be within its angle of influence. Angle of influence of all nodes is a part of angle of influence of their respective parents. Each node gets part of the angle proportional to the number of nodes of the tree rooted in that node. For details of the distribution see [4]. The purpose of the angle of influence is to make paths from root to leaves follow a certain direction (they should not zig-zag too much). The limited angle may force us to increase radius of a layer so that the children fit into the angle (if the angle is constant and the radius increases the actual space available to the nodes increases).

After all nodes are placed, the edges are drawn between them. There are two types of edges – tree edges and non-tree edges. The tree edges can be drawn as simple straight lines connecting the nodes, because they only connect nodes on adjacent layers and furthermore the children are in similar position on their layer as is the parent on the previous layer (thanks to the angle of influence). The non-tree edges are more complex as they can connect very distant nodes. We draw them by starting with a straight line and adding bends to the line so that it avoids all nodes in its path. Details are described in [5].

An example of visualized RDF data is shown in Figure 1.

3 Navigation

We need to provide the user not only with well-arranged drawing of the graph but also with means to alter the view of graph to match his or her needs. As the basic operation, we decided to allow the user to extend the view by adding a neighbor of an already displayed node. This way we get the navigation tree, which is the main structure we visualize. Beside this operation we also enable the user to reduce the view by a single node or some connected subtree. The user can decide that the current view of the graph is no longer interesting but he or she still wants to continue with the navigation. For this purpose we proposed support view restructuring operations.

In all these operations we preserve the mental map of the user as much as possible. The mental map preservation, i.e. the stability of the layouts, is a key issue in dynamic graph drawing. The quality of the layout can be evaluated by measuring the movement of the nodes between successive layouts. The movement should be small, especially in the areas of the graph unchanged by the navigation.

In [7] authors identified some "general factors" that contribute to the mental map preservation.

Predictability – the aim is to make the jump from one view to the next one predictable.

Degree of change – the jump itself should be minimized as well. The new view should preserve the node coordinates as much as possible. In a strict model, all nodes not involved in the modification of the view have to preserve their relative order in both coordinates.

Traceability – the user should be able to notice the changes as they take place, so that they can be integrated in his or hers mental map. This is usually achieved via animation. However, not every animation gives good support for the mental map preservation. In [8] the criteria for a good animation were sketched. In our animations we fulfill these criteria to get easily traceable animation process. Note that only the navigation tree is animated, since the non-tree edges may completely change their position and may lead to too complex animation, which makes the whole transition less comprehensible to the user.

3.1 View expansion

Let $G = (V, E)$ be the currently displayed graph and $T = (V, E')$ the navigation tree. Then $E' \subseteq E$ holds. The user selected node $v \in V$ with a set of neighbors $N(v)$. If $N(v) \subseteq V$ then all neighbors of v are already displayed and v can not be used for further navigation. In the other case, let $N'(v) = N(v) \setminus V$ be the set of yet undisplayed neighbors of v and let $u \in N'(v)$. Then we can extend the displayed graph and navigation tree like this:

- The set of nodes is extended by the node u.
- The set of edges is extended by all edges between u and the already displayed nodes.
- The edge (u, v) is added to the navigation tree.

Drawing of the new view can be done in several different ways. The most simple one is to completely redraw the whole graph. This way we use the least space to draw the new view.

However, we enhance the mental map preservation, if we do not redraw the whole graph. Sometimes, only small (or even no) change of displayed nodes coordinates is enough. This occurs for example if a node fits into the angle of influence of its predecessor. In this case we can only insert the node.

Therefore, we use the algorithm $UpdateInsert$ instead of redrawing the whole tree. We suppose that the node v_k is inserted and $r_T = v_0, v_1 \dots v_k$ is the path from the root of the navigation tree to the node v_k.

How deep we need to recompute the coordinates in the navigation tree? We get the answer from the function $Insert$. The main idea is following: If a child can not fit into the parent's angle of influence, it may be possible, that he has too many children and too little space. So it may ask one of its predecessors (first its direct predecessor and then further towards the root) to recompute their angles of influence according to actual situation. To achieve a correct distribution of angles of influence, the first son of each node gets the whole angle and every further son gets zero angle of influence. This is of course recomputed in the case that he gets a new descendant.

```
INSERT($v_k$)
 1  $i \leftarrow k$
 2  while $i > 0$
 3     do if $v_k$ fits into angle of influence of $v_{k-1}$
 4           then if $v_k$ $is\ first\ ancestor$
 5                   then $\alpha(v_k) = \alpha(v_{k-1})$ // gets whole angle
 6                   else $\alpha(v_k) = 0$ // gets zero angle
 7                return $(i, v_i)$
 8           else redistribute angles of influence in $T(v_{i-1})$
 9                $i \leftarrow i - 1$
10  return $(1, r_T)$
```

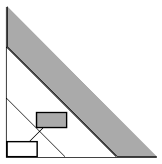
After we know that only subtree $T(v)$ will be influenced we recompute the coordinates (the angle γ) in this subtree (using function $Recompute$). It is possible, that the radii of some layers are affected too (they may increase). The coordinates of nodes outside the subtree $T(v)$ are not recomputed, they are only shifted further from the root (if their layer radius was increased).

```
RECOMPUTE($j, T$)
1  for each $h$ in $\{j, j+1, \dots\}$
2     do COMPUTE($r_h$)
3        for each $v$ in $L(h-1) \cap T$
4           do // v has children $v_1 \dots v_k$
5              for $i = 1$ to $k$
6                 do COMPUTE($\alpha(v_i)$)
7                    COMPUTE($\gamma(v_i)$)
```

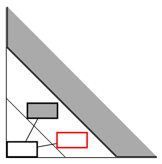
The function $UpdateInsert$, that utilizes the previous functions, can be drafted as follows. Note, that although we recompute coordinates of all nodes on layers $l \dots max\_layers$ they only shift away from the root ($\gamma(v)$ is unchanged).

```
UPDATEINSERT($v_k$)
1  $(l, v) \leftarrow Insert(v_k)$
2  $Recompute(l, T(v))$
3  for $j = l$ to $max\_layers$
4     do for each $v$ in $L(j)$
5           do $y(v) \leftarrow r_j \cdot \frac{sin\ \gamma(v)}{sin\ \gamma(v) + cos\ \gamma(v)}$
6              $x(v) \leftarrow r_j - y(v)$
```

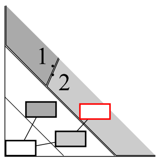
Example At the beginning the tree consists of only a root r_T. Then we add a node v_1, which is the first son of the root. So the situation is the following:



Both the root and its son get the whole angle $\langle 0, 90 \rangle$ as their angle of influence. Now, we add a second son v_2 to the root. As written before, the second son gets zero angle of influence. After this step, the first son v_1 still has the whole layer as its angle of influence.



Now the second son gets a child u. It can be easily seen that it can not fit into the zero angle of influence. So the angles of influence have to be recomputed. The tree $T(v_1)$ rooted in v_1 has only one element and the tree $T(v_2)$ rooted in v_2 has two nodes. According to the distribution function, the angle of influence of the root is divided to its children in proportion to the size of the trees rooted in them. So the node v_1 gets one third of the layer and the node v_2 gets the rest.



Now, the node u fits into the angle of influence of its father. Should this not be the case, the whole tree would have be recomputed with new layer radii.

Animation In [8] authors drafted four steps of a good animation process. The animation of the expansion of the navigation tree is divided into two steps (steps 3 an 4 of the suggested process):

1. The linear transition stage – is for easier comprehension divided into two parts
   (a) Increasing radii of all layers, which were affected by the expansion (we do this first, since the nodes may not fit into the smaller layers). This step is only 'scaling' of the image. All nodes are moved along the line connecting them to the coordinates origin away from the root.
   (b) Now, the nodes of the tree $T(v)$ (contained in the previous view) are moved along the layers they are positioned on.
2. Show newly added elements (the new node v_k and the edge to its predecessor).

3.2 View reduction

It is possible that the user will consider some subtrees of the navigation tree to be no longer interesting. For this reason we allow the view to be reduced by removing a node or a whole subtree.

Let $G = (V, E)$ be the currently displayed graph and $T = (V, E')$ the navigation tree. If the user selects the node $v \in V$, we reduce the displayed graph and the navigation tree by removing the subtree $T(v) = (V_v, E_v)$ – the subtree rooted in v – like this:

- The set of nodes is reduced by the set V_v.
- The set of edges is reduced by all edges connecting V_v and other displayed nodes.
- The edges from E_v and the edge between v and its predecessor (v_{pred}) is removed from the navigation tree.

Like in the case of the view expansion, there is the possibility to completely redraw the graph or to only do some update. As the $UpdateInsert$ algorithm can also use the space released by removing some nodes, we use a very easy alternative, where the nodes that remain displayed retain their original coordinates. Even though the graph is not redrawn, we get some free space. There are two possibilities, how to manage this free space:

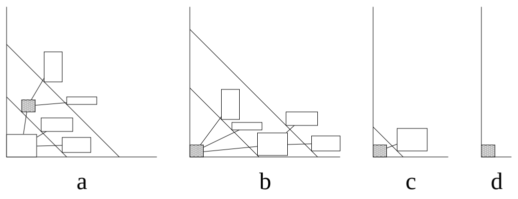
- Completely ignore the free space and wait for $InsertUpdate$ to manage it.
- Distribute the free space (according to the distribution function) among the neighboring siblings of the removed node (as you can see in Fig. 2(c)). The space can be directly used for new children of this siblings.

Animation In the case of view reduction, the animation is unnecessary, since all nodes retain their coordinates.

3.3 Restructuring the view

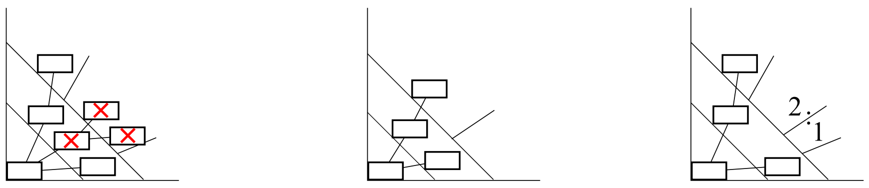
While navigating the graph, the user can decide that the current view of the graph is no longer interesting but he or she still wants to continue with the navigation. In that case we allow the user to create a new view of the graph using one of this options:

- **Selection of new origin** (see case (b)) keeps all displayed nodes. Only the navigation tree is altered by selecting different node as the root of the tree.
- **Preservation of path** to a selected node (see case (c)) removes all displayed nodes except for the path from the root to the selected node. The selected node is the new root of the navigation tree.
- **Preservation of node** (see case (d)) is the most extreme change of the view. Only the selected node is retained (of course in the form of merged node) and it is set as a new root of the navigation tree.



Animation Just like view expansion and reduction, the view restructuring operations are animated. If the users wants to preserve only one node, all other nodes and edges vanish and the remaining node moves to the origin of the coordinate system.

When the whole path is preserved, all nodes and edges outside the path vanish, the path rotates around its center. Then all nodes and edges move to their target locations.



(a) Before reduction (b) Redrawing (c) Lazy reduction (distr. of the free space)

Figure 2: View reduction

The selection of a new origin is the most complex case. All of the nodes and edges of the graph are preserved and all of them have to be moved to new locations. This animation is split into several steps, where in each step the nodes move only to neighboring layers.

To make each step easy to follow and preserve the user's mental map as much as possible, we would like to preserve order of the nodes within a layer. To be more exact, for every pair of nodes u and v, if the path from the node u to the root is above or below the path from v to the root before a step of the animation, then they must have the same relative position after the step. If the paths have a common part they are considered to be both above and below each other and their actual relative position is determined by the disjoint parts of the paths if there are any. If there are no disjoint nodes the paths are considered to be both above and below each other (this occurs if u is a descendant of v or v a descendant of u) and their position after the step can be either of the two.

If we maintain this condition, the user's mental map should be relatively well preserved and, furthermore, the moving nodes would not extensively cross each other.

In each step, let r_1 be the current root of the tree, r_2 the node user selected as the new root, P path from r_2 to r_1 minus r_1, and $p \in P$ a child of r_1 (p is the node of P that is closest to the root). For a node v we denote $T(v)$ to be the tree rooted in v.

One step of the animation consist of simple linear translation of nodes. But there is one operation that is similar to rotation known from e.g. AVL-trees. It is the transformation of the root and its children which can be performed in two directions. The direction is chosen at the start of view restructuring and does not change between the steps. The direction is based on the position of r_2 – if the center of r_2 is above the line going in the angle of 45 degrees from the coordinate origin the rotations is done counter clockwise. If the node is below the line, it is clockwise. The following text describes the counter-clockwise variant.

All nodes from $T(p)$ are moved one level closer to the root, making p the new root of the tree. The r_1 node is moved one layer in the opposite direction (making r_1 a child of p – rotating them), along with all of its children that were position below p. The trees rooted in these children are moved as well. The r_1 node and the children are placed below P, thus preserving the order of nodes within layers as was defined earlier.

There is a problem with children of r_1 (and subtrees rooted in those nodes) that were above the node p. If we wanted to preserve order of nodes on each layer, then those nodes have to be above P. But they are descendants of r_1, which was moved below P, and so all of its descendants must be below P because tree edges never cross. This is a contradiction and so the order of the nodes cannot be maintained when our algorithm is used. It is relatively easy to find an example which demonstrates that for some graphs, the condition cannot be maintained by any algorithm.

This means we have to violate the condition in some way, which would very likely result in some nodes crossing other nodes during the animation. But we still may maintain the condition to certain degree – to make it hold at least for the descendants of the root and subtrees rooted in the descendants. This way the mental map of the user is at least partially preserved. As for the crossings we have two options. We can either completely avoid them or try to minimize their impact. We came up with several ways of avoiding the crossings but they all result in an animation that is too complex and hard to follow.

For this reason, we decided to simply swap the positions of the problematic nodes and the $T(p)$ using the basic linear animation, which inevitably results in one nodes crossing the others on their way. A slightly more sophisticated animation could avoid that – by increasing radii of the layers we can make sure nodes from one group pass through spaces between the nodes of the other group. But again, the resulting animation would be too complex and not fluent, making the very basic linear animation a better choice.

Still, there is something that can be done to make the swapping of the nodes look better. The idea is to use the z-axis as well, by making one set of nodes appear to get closer to the user and the other farther from user. Unfortunately, there is no easy way to achieve this in current SDL-based implementation which is one of the reasons the visualizer is being rewritten for the WPF.

4 Existing approaches to visual navigation in RDF

Since RDF data have been around for quite some time, there are already several tools that try to visualize it. Many of them display the whole graph, which is not suitable for large data because it requires too much resources and the resulting view is not clear. The visual navigation is impossible in this case. However, there are still some other tools that also allow the visual navigation in the RDF data.

- **Node-centric RDF Graph Visualization** [10] is one of the few tools that do not try to display the RDF graph precisely. According to the user's choice, it always displays tree of node's ancestors and descendants. If any of them can be reached by more than one path (which would create a non-tree edge in our solution) the node is displayed multiple times (once for each path) which preserves the tree structure of descendants and ancestors. One disadvantage of this tool is that it only displays two levels of ancestors and descendants and does not try to handle nodes with high degree.
- **Paged Graph Visualization (PGV)** [3] is similar to our tool because it does not try to display the whole graph. Incremental algorithm is used to layout the explored sub-graphs. However, the expansion of the view is the only navigation operation in the PGV explorer. Moreover, the expansion extends the view by all neighbors of selected node. It can be problem mainly with nodes that have high degree. Even though authors claim that the Ferris-Wheel technique handles high-degree nodes (but only with nodes having at most hundreds of neighbors) the expansion of all neighbors of such node is too space consuming, especially if no reduction of the view is allowed.
- **IsaVis** [9] is a visual environment for browsing and authoring RDF models, represented as directed graphs. The graph is visualized once and can be explored using cameras that can be moved and zoomed. The user can not reduce or modify the view and can only browse the whole graph. If the graph is too big even the zooming may not lead to easily readable view.

Despite significant diversity of available visualization tools for RDF data (for more details about visualization tools see [4]), only a few of them can be used to visualize large data. None of the tools uses technique similar to node merging and all of them run into trouble when the data contain nodes with very high degree although such nodes can commonly be found in the real world data. In most of the systems, the navigation in the graph is not enabled or is limited to browsing through the whole displayed graph.

5 Conclusions

For our RDF visualizer, we have developed navigation techniques that we believe are as much user-friendly as possible. We extensively use animations to help us with this. We have created a working implementation using the Trisolda semantic web infrastructure [6].

In the future, we would like to study the problem of non-tree edge animation in greater detail. Although in general the animation would be impossible for the user to follow, if the change was small enough, it might be helpful.

References

[1] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, May 2001.

[2] J. J. Carroll and G. Klyne. Resource Description Framework: Concepts and Abstract Syntax. W3C Recommendation, 2004.

[3] L. Deligiannidis, K. J. Kochut and A. P. Sheth. RDF data exploration and visualization. *CIMS '07: Proceedings of the ACM first workshop on CyberInfrastructure*, 39–46, ACM, New York, 2007.

[4] J. Dokulil and J. Katreniaková. Visual Exploration of RDF Data. *In: SOFSEM 2008: Theory and Practice of Computer Science*, 672–683 Springer Berlin / Heidelberg, 2008.

[5] J. Dokulil and J. Katreniaková. Drawing of edges in RDF visualization. Technical report, Faculty of Mathematics, Physics and Informatics, Comenius University, Bratislava, 2008.

[6] J. Dokulil, J. Tykal, J. Yaghob, and F. Zavoral. Semantic web infrastructure. In *First IEEE International Conference on Semantic Computing*, 209–215, Los Alamitos, California, 2007. IEEE Computer Society.

[7] M. Freire and P. Rodríguez. Preserving the mental map in interactive graph interfaces. In *Proceedings of Advanced Visual Interfaces (AVI 2006)*, 2006.

[8] C. Friedrich and P. Eades. Graph Drawing in Motion. *Journal of Graph Algorithms and Applications*, 6(3):353–370, 2002.

[9] E. Pietriga. IsaViz: a Visual Environment for Browsing and Authoring RDF Models. *WWW 2002, the 11th World Wide Web Conference*, Honolulu, USA, May 2002. http://www.w3.org/2001/11/IsaViz/.

[10] C. Sayers. Node-centric RDF Graph Visualization. Technical report HPL-2004-60, HP Laboratories Palo Alto, April 2004.