


# Path Patterns Visualization in Semantic Graphs

**José Paulo Leal**

CRACS & INESC-Porto LA

Faculty of Sciences, University of Porto, Portugal

zp@dcc.fc.up.pt

 <https://orcid.org/0000-0002-8409-0300>

---

## Abstract

Graphs with a large number of nodes and edges are difficult to visualize. Semantic graphs add to the challenge since their nodes and edges have types and this information must be mirrored in the visualization. A common approach to cope with this difficulty is to omit certain nodes and edges, displaying sub-graphs of smaller size. However, other transformations can be used to abstract semantic graphs and this research explores a particular one, both to reduce the graph's size and to focus on its path patterns.

Antigraphs are a novel kind of graph designed to highlight path patterns using this kind of abstraction. They are composed of antinodes connected by antiedges, and these reflect respectively edges and nodes of the semantic graph. The prefix “anti” refers to this inversion of the nature of the main graph constituents.

Antigraphs trade the visualization of nodes and edges by the visualization of graph path patterns involving typed edges. Thus, they are targeted to users that require a deep understanding of the semantic graph it represents, in particular of its path patterns, rather than to users wanting to browse the semantic graph's content. Antigraphs help programmers querying the semantic graph or designers of semantic measures interested in using it as a semantic proxy. Hence, antigraphs are not expected to compete with other forms of semantic graph visualization but rather to be used a complementary tool.

This paper provides a precise definition both of antigraphs and of the mapping of semantic graphs into antigraphs. Their visualization is obtained with antigraphs diagrams. A web application to visualize and interact with these diagrams was implemented to validate the proposed approach. Diagrams of well-known semantic graphs are also presented and discussed.

**2012 ACM Subject Classification** Computing methodologies → Semantic networks, Human-centered computing → Graph drawings

**Keywords and phrases** semantic graph visualization, linked data visualization, path pattern discovery, semantic graph transformation

**Digital Object Identifier** 10.4230/OASIS.SLATE.2018.15

**Funding** This work is partially funded by the ERDF through the COMPETE 2020 Programme within project POCI-01-0145-FEDER-006961, by National Funds through the FCT as part of project UID/EEA/50014/2013, and by FourEyes. FourEyes is a Research Line within project “TEC4Growth – Pervasive Intelligence, Enhancers and Proofs of Concept with Industrial Impact /NORTE-01-0145-FEDER-000020” financed by the North Portugal Regional Operational Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement, and through the European Regional Development Fund (ERDF).



© José Paulo Leal;

licensed under Creative Commons License CC-BY

7th Symposium on Languages, Applications and Technologies (SLATE 2018).

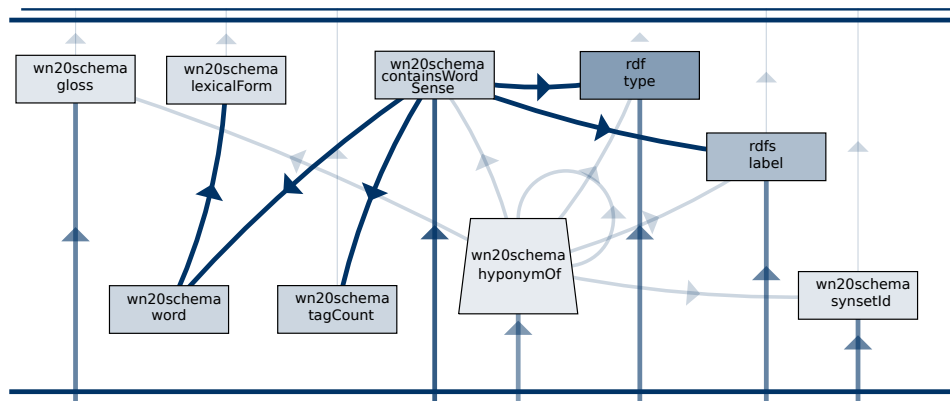
Editors: Pedro Rangel Henriques, José Paulo Leal, António Leitão, and Xavier Gómez Guinovart

Article No. 15; pp. 15:1–15:15



Open Access Series in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany



■ **Figure 1** An antigraph diagram of WordNet 2.1.

## 1 Introduction

Graphs, like most mathematical entities, are inherently visual. In fact, our mathematical intuition relies heavily on our ability to visualize angles, functions, vectors or geometric figures. It fails us, for instance, when we try to visualize a hypercube. However, the projection of multidimensional solids in 3 or 2-dimensional spaces give us an idea of these entities' shape. The visualization of the hypercube is an apt metaphor of the key insight that drives this research: the understanding of a complex entity may be improved by looking at the shadow it casts.

Graphs have a particular role in visualization since they are the data model of most diagrams. Diagrams enrich graphs in two ways: a graphical syntax for nodes and edges; and the layout of nodes and edges on a surface. Different kinds of diagrams have been developed for many purposes. These diagrammatic languages are used for modeling and visualizing relationships among entities, even when they are purely abstract.

In spite of their ability to show relationships and symmetries, large diagrams are difficult to visualize. Too many nodes and too many entangled edges reduce our perception on the underlying graph. This is particularly the case of the semantic web, where graphs are growing increasingly larger and denser. Section 2 presents different attempts to provide visualizations of large semantic graphs, with efficient approaches to process large quantities of data and methods to abstract them, mostly by omitting nodes and edges with certain features. These diagrams represent the graphs themselves, and the layout may highlight general properties, such as symmetry, but usually they do not reveal specific features such as patterns formed by nodes and edges.

The novelty of the antigraph approach is the abstraction of large semantic graphs into a much smaller graph highlighting its *path* patterns. The abstraction process consist on mapping semantic graphs into antigraphs, a particular kind of graph with an associated diagram type. Sets of edges with the same type are mapped into nodes and sets of nodes into edges. This process reverses the nature of the constituents of a graph, thus the metaphor of antimatter where positrons, rather than electrons, revolve around a nucleus made of antiprotons and antineutrons, rather than the protons and neutrons of regular matter. Section 3 defines antigraphs and their relationship with semantic graphs. It also introduces the antigraph diagrams used for their visualization and provides small examples of this kind of diagram. The final subsection compares antigraphs with other forms of representing the properties of semantic graphs, such as ontologies.

An implementation of the mapping and diagram layout is described in Section 4. It is deployed as a web application for browsing antigraphs and exporting them as vector images, such as the diagram representing Wordnet 2.1, shown in Figure 1. It is available online<sup>1</sup> and is useful to validate the proposed approach. Section 5 presents examples of diagrams produced with this tool to illustrate different techniques to create meaningful visualizations of large semantic graphs and discover relevant path patterns. The last section summarizes the research presented in this paper, highlights its main contributions and identifies opportunities for future research.

## 2 Related Work

Knowledge bases such as WordNet<sup>2</sup> [8], Yago<sup>3</sup> [13] and DBpedia<sup>4</sup> [4], have a massive amount of information. A typical representation of these knowledge bases are node-link multigraphs, where each node has a type and nodes are connected by links representing the relationship between them.

A convenient way to analyze this data is using data visualization. The most common type of visualization is focused on the analyzes of resources, in particular, those with a high outdegree. The main challenge of semantic graph visualization and management is related to the graph size. This type of graphs has several thousands of nodes and edges and are usually very dense.

The literature presents several approaches to handle the visualization and management of node-link graphs. Most of the related work on massive graphs visualization is handled through hierarchical visualization. This type of approach has low memory requirements, however, it depends on the characteristics of the graph. The graph hierarchy can be extracted using different kinds of methods. Tools such as ASK-GraphView [1], Tulip [3] and Gephi [5] explore clustering and partitioning methods, creating an abstraction of the original graph that can be easily visualized. Another technique used to build hierarchies is based on the combination of edge accumulation with density-based node aggregation [17]. Visual complexity can also be reduced by hub-based hierarchies, where the graph is fragmented into smaller components, containing many nodes and edges, making meta nodes, as described in [15]. GrouseFlocks [2] allows users to manually define their own hierarchies.

There are specific tools when the semantic graph is in Resource Description Framework (RDF) format, however, they require loading the full graph. Some desktop-based tools, such as Protégé<sup>5</sup> and RDF Gravity<sup>6</sup>, are mainly used with purpose of aiding developers to construct their ontologies, providing also complex graph visualizations. Of all available tools for linked data visualization the most notable ones are the following. Fenfire [11] is a generic RDF browser and editor that provides a conventional graph representation of the RDF model. The visualization is scalable by focusing on one central node and its surroundings. RelFinder<sup>7</sup> [12] is a tool that extracts from a Linked Open Data (LOD) source the graph of the relationships between two subjects. It provides an interactive visualization by supporting systematic analysis of the relationships, such as highlighting, previewing and filtering features.

---

<sup>1</sup> <http://quilter.dcc.fc.up.pt/antigraph>

<sup>2</sup> <https://wordnet.princeton.edu/>

<sup>3</sup> <https://www.mpi-inf.mpg.de/yago-naga/yago>

<sup>4</sup> <http://wiki.dbpedia.org/>

<sup>5</sup> <http://protege.stanford.edu/>

<sup>6</sup> <http://semweb.salzburgresearch.at/apps/rdf-gravity/>

<sup>7</sup> <http://www.visualdataweb.org/refinder.php>

ZoomRDF [16] is a framework for RDF data visualization and navigation that uses three special features to support large scale graphs. It uses *space-optimized* visualization algorithms that display data as a node-link diagram using all visual space available. *Fish-eye zooming* is another feature that allows the exploration of selected elements details, while providing the global context. The last feature is the *Semantic Degree of Interest* assigned to all resources that consider both the relevance of data and user interactions. LODeX [6] produces a high-level summarization of a LOD source and its inferred schema using SPARQL endpoints. The representative summary is both visual and navigable. The platform graphVizdb<sup>8</sup> [7] is a tool for efficient visualization and graph exploration. It is based on a *spatial-oriented* approach that uses a disk-based implementation to support interactions with the graph.

### 3 Antigraph definition

The most distinctive feature of antigraphs is that nodes and edges are reversed relatively to the semantic graphs that generated them. Subsection 3.1 explains the motivation behind this decision and characterizes the main components of antigraphs, namely antinodes and antiedges, as well as their features.

The proposed approach to the visualization of semantic graphs can be divided into two parts. Firstly, the semantic graph is abstracted to another graph – the *antigraph* – that promotes types of edges. Secondly, this abstracted graph is visualized using a special kind of diagram – the *antigraph diagram* – that emphasises path patterns. The following two subsections detail each facet of the antigraph approach.

Finally, Subsection 3.3 discusses antigraphs as abstractions of semantics graphs, in relations to ontologies.

#### 3.1 Motivation

Nodes have the main role in a graph. Edges connect nodes and establish relationships among them. The goal of antigraphs is to abstract a given graph, highlighting edges and reducing its size. Hence, in an antigraph nodes and edges are reversed, i.e. an *antinode* abstracts edges and an *antiedge* abstracts nodes. A graph and its *antigraph* have the same duality of matter and antimatter (where electrons are replaced by positrons and, protons by antiprotons and neutrons by antineutrons).

It is important to note that an antinode abstracts an edge *type* rather than a single edge. Hence the order (the number of nodes) of an antigraph is in general much smaller than that of the graph it abstracts. For instance, the graph of WordNet 2.1 has about 2 million edges with 27 edge types, hence 27 is the order of the of reductions that abstract it.

An antiedge expresses a relationship between a pair of antinodes, namely that the edge types it represents can be connected to form a length 2 path. Two edges form a length 2 path when the target of the first is the source of the second. Since an antiedge represents a set of nodes, the size (the number of edges) of an antigraph is much smaller than the size of the graph it abstracts. Considering that antiedges can be laces, the number antiedges is less or equal to  $n^2$ , where  $n$  is the number of antinodes. For instance, the size of the WordNet 2.1 graph is about half a million but the size of its antigraph is only 214, well below the maximum of  $27^2 = 729$ .

---

<sup>8</sup> <http://graphvizdb.imis.athena-innovation.gr/>

The expressiveness of antinodes and antiedges is increased by adding weights to them. The weight of an antinode is the percentage of edges with the type it abstracts. For instance, if a graph has half of its edges of type  $t$  then the antinode reflecting  $t$  has weight  $1/2$ . Hence, antinodes with higher weight reflect edge types that are more frequent in the graph. Obviously, the sum of antinode weights must be 1.

By the same token, the weight of an antiedge is the percentage of nodes that participate in length 2 paths involving edge types they have as source and target, respectively. For instance, if an antinode reflects the edge type  $t_1$  and another the edge type  $t_2$ , and  $1/3$  of the nodes are target of  $t_1$  and source of  $t_2$ , then the weight of the antiedge  $t_1 \rightarrow t_2$  is  $1/3$ .

One would expect every node to be reflected by an antiedge, but for that to happen the nodes that are just sources (not the target of any edge) or just targets (not the source of any edge) must also be abstracted by antiedges. To ensure that all nodes are reflected by antiedges it is necessary to introduce two special antinodes: *bottom*, denoted as  $\perp$ ; and *top*, denoted by  $\top$ . The bottom antinode represents a nonexisting edge type that would come before the start of a path. Conversely, the top antinode represents a nonexisting edge type that would come after the end of a path. Both special antinodes have weight 0, thus maintaining the invariant that the sum of all weights is 1.

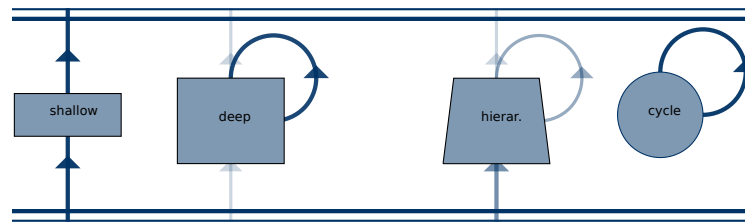
The two special antinodes – bottom and top – allow the definition of antiedges that abstract nodes that are only source or target of edges. These antinodes are considered *special* to differentiate them from *regular* antinodes, that have an associated edge type. The antiedge  $\perp \rightarrow t$  abstracts the nodes with a null indegree that are sources of edges with type  $t$ , and the antiedge  $t \rightarrow \top$  abstracts the nodes with a null outdegree that are targeted by edges of type  $t$ .

In fact, the in(out)degrees of nodes must be taken into consideration in the weight of all antiedges. Consider a node  $n$  with indegree 2 and outdegree 3. For instance, if the two incoming edges and the three outgoing are of different types then the contribution of that node to each antiedge is  $1/6$ . Thus, the weight of an antiedge is the percentage of connecting nodes in paths formed by the edge types, pondered by their in(out)degrees. With this definition, the sum of antiedges weights is also 1.

As explained above, the introduction of the special antinodes bottom and top is essential to abstract all the nodes of the original graph in antiedges connecting them. One may wonder what other antinodes types should be considered. It should be noted that antinodes may have antiedge laces if the graph contains homogeneous paths, i.e. paths formed by a single type of edge. Since the goal of antigraphs is to highlight path patterns, it is important to distinguish different cases that would be amalgamated by generic antinodes with laces.

Certainly, not all antinodes have laces. These are considered *shallow* antinodes since they have at most paths of length 1. In contrast a *deep* antinode has homogeneous paths of higher length through its lace. Special cases of deep antinodes can be also considered: *cyclical*, where the laces contain homogeneous cycles, i.e. cycles using only the type of edge represented by the antinode; and *hierarchical*, where the laces represent confluent paths, i.e. where the nodes in homogeneous paths have branching factor above 2. These types provide information on the kind of paths formed “within” an antinode, similar to the information that can be extracted from other antiedges relating different antinodes.

In summary, an antigraph is an abstraction of a semantic graph. This does *not* mean that an antigraph is a sort of schema. A semantic graph does not comply with its antigraph, its the other way round: antigraphs have a functional dependency to semantic graphs. Thus, the information provided by an antigraph is of a different nature of that of an RDF or OWL ontology. This point is analyzed in greater detail in Subsection 3.3, after formalizing the definitions of antigraph and antigraph diagram.



■ **Figure 2** Catalog of antinode types.

### 3.2 Diagram language

As explained in the previous subsection, an antigraph is an abstraction of a semantic graph. The antigraph diagram language is a visual representation of an antigraph intended to highlight the path patterns of the abstracted semantic graph. An antigraph has antinodes of different types connected by antiedges.

The type of an antinode is conveyed by its shape. A shallow antinode is represented by a horizontal rectangle, while a deep antinode is represented by a vertical rectangle. A cyclical antinode is represented by a circle or an ellipse, and a hierarchical antinode is represented by an isosceles trapezoid. The position of these shapes is their geometric center. The antigraph depicted in Figure 2 is a sort of catalog of antinode types, where the label of each regular antinode is the type's name.

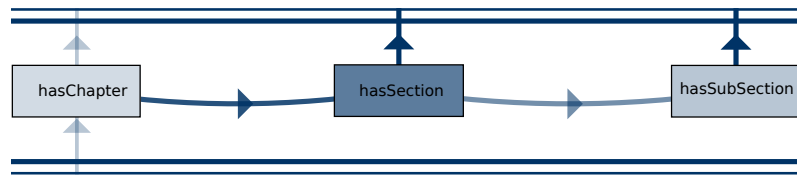
The bottom and top antinodes are represented by a pair of parallel lines rather than shapes. As can be seen also in Figure 2, the parallel lines that represent each of these antinodes have different widths. The bottom antinode has a larger upper line and the top antinode is the reverse. The bottom and top antinodes are located respectively at the bottom and top of the diagram, as their names suggest. This way the paths created by antiedges tend to be directed upwards.

Unlike antinodes, antiedges have a single type. Hence, they are represented all by solid lines with an arrowhead positioned in their middle pointing to the target. Lines connecting from the bottom antinode, or to top antinode, are straight. All the others are curved so that antiedges with opposite directions do not overlap.

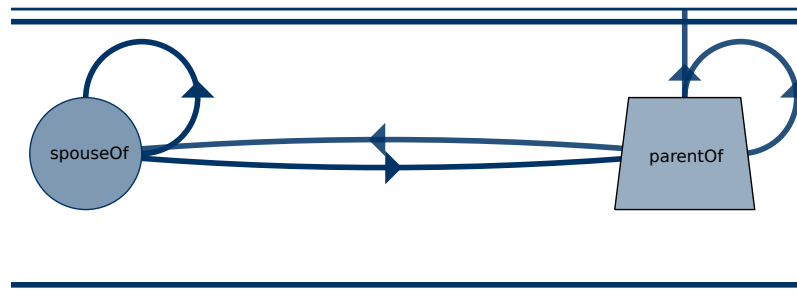
Antinodes and antiedges have weights in the  $[0, 1]$  interval. Actually, both regular antinodes and antiedges have always nonnull weights; special antinodes (top and bottom) have null weights by definition. The nonnull weights of regular antinodes and antiedges are conveyed graphically too. The weight of an antinode is shown as a transparency, making dimmer the antiedges representing a smaller number of edges in the abstracted graph. The same principle applies to weights of antiedges. In this case, the weight is also shown as line width, making thicker the antiedges that represent a larger number of nodes. The semantic graph that originated the antigraph in Figure 2 has all edge types with the same number of edges, hence all antinodes have the same weight, thus they all have the same shade. A different thing happens with antiedges; each has a different shade, reflecting their different weights.

The regular antinodes in the catalog diagram are not connected to each other, just to bottom and top (with the exception of the cycle). This means that they do not form “joins”. Using a syntax borrowed from SPARQL, it can be said that the semantic graph that generated it lacks triple patterns of the form

$$\begin{aligned} ?a \ ?p \ ?b \ . \\ ?b \ ?q \ ?c \ . \end{aligned}$$



■ **Figure 3** Book structure.



■ **Figure 4** Family relationships.

The example in Figure 3 represents the structure of books, where a book has chapters and these have sections. The antigraph of such semantic graph has the properties *hasChapter*, *hasSection* and *hasSubSection*.

In this case “joins” are created using multiple edge types hence the antinodes have antiedges connecting them. In particular *hasChapter* is connected to *hasSection* and this to *hasSubSection*. The reader should note that the three regular antinodes are connected to the top, meaning that there are chapters, sections and subsections that are not subdivided, and that only *hasChapter* is connected from bottom, meaning that only these are connected from root elements of the hierarchy.

The previous example reflects a hierarchical structure, although with a different type of edge for each layer. The example in Figure 4 reflects a semantic graph with a couple of family relationships, namely *spouseOf* and *parentOf*. Their associated antinodes both have laces, which means that paths with a single type of edges can be created. The *parentOf* antinode has hierarchic as type, meaning that paths of length greater than 3 can be created and has an average branching factor above 2.

The simple patterns identified in the small examples above occur also in larger semantic graphs. Section 4 presents an antigraph browser that allows us to discover combinations of these patterns in in larger examples, as those analyzed in Section 5.

### 3.3 Antigraphs and ontologies

Semantic graphs are frequently encoded as sets of triples in the Resource Description Framework (RDF). This framework supports multiple vocabularies, including a vocabulary to describe other vocabularies – RDF Schema (RDFS) – which in turn lays the foundations for a richer ontological language – OWL. RDFS and OWL describe vocabularies in terms of classes and properties, where classes provide types for nodes and properties types for edges of semantic graphs, and define hierarchical relationships among those types.

The definition of semantic graph on which the definition of antigraphs relies is also based on types. However, these types are of a different nature. These node and edge types are not RDFS or OWL classes and properties, and they are not hierarchically related among



themselves. The node and edge types in the definition of antigraphs are the actual URIs used to label them.

Nevertheless, ontologies and antigraphs are somehow related in the sense that they both abstract semantic graphs. Thus, it is relevant to question if these two concepts – ontologies and antigraphs – overlap or compete in any way.

The concept of ontology varies for different communities [9]. In the semantic web, an ontology is usually understood as a formal definition of a domain of discourse. It declares a taxonomy of concepts and relationships among them. For instance, an ontology may declare *cat* and *dog* as classes, both as subclasses of *pet*, and the property *hasName* associating pets to their names (strings). RDFS and OWL ontologies are themselves RDF graphs, although not all RDF graphs are ontologies. In fact, most RDF graphs assert facts on resources using types and properties, such as “Rex is a dog”<sup>9</sup>, but they do not define hierarchies of classes (concepts) and properties (relationships).

By using inference with an ontology it is possible to entail new facts from existing ones, such as “Rex is a pet”. The reverse, to induce an ontology from a collection of facts, is much more complex. It is possible to process statements such as “Rex is a dog” and “Fifi is a cat”, “Rex is a pet” and “Fifi is a pet” and induce an ontology similar to the example in the previous paragraph. However, ontologies are not usually created this way.

Ontologies prescribe how certain semantic graphs must be. They are not a sort of a “summarization” of existing semantic graphs. If an ontology is applicable to a particular semantic graph then the facts of the later should be consistent with the former; and as more facts are added, that consistency should be preserved without changing the ontology.

An antigraph is, in fact, a summarization of a semantic graph. It maps edges into antinodes and nodes into antiedges in a way that the antigraph paths condense several paths of the semantic graph it abstracts. However, only paths that actually exist in the semantic graph are abstracted into antigraph paths, not all the paths that would be consistent with the ontology. Moreover, since antinodes and antiedges have weights, the path frequency is also presented by the antigraph, which has no parallel in ontologies. As a semantic graph evolves and new nodes and edges are added (or removed), its antigraph may change to reflect it. In some cases, only the weights will be affected, if no kinds of path are created. In other cases, new antinodes result from edge types that did not exist before.

In summary, antigraphs and ontologies are different kinds of abstractions. Antigraphs abstract paths, highlighting the most frequent ones. Ontologies abstract relationships among concepts. The two abstractions are non-overlapping and are in fact complementary.

## 4 Antigraph browser

This section describes the design and implementation of a web application developed to validate the concept of antigraph. This web application – the antigraph browser – produces interactive antigraph diagrams from several data sources and is freely available online<sup>10</sup>.

The antigraph browser is a Java web application developed with the Google Web Toolkit (GWT). It is composed of a client front-end running on a web browser and a server back end. The server is responsible for transforming a semantic graph in RDF format into an antigraph that is sent to the client. The front end is responsible for laying out diagrams and managing user interaction, as explained in the following subsections.

<sup>9</sup> “Rex is a dog” are two RDF facts. Assuming *ex* as an alias for a namespace, that sentence would be represented by the RDF facts «*ex:rex ex:hasName "Rex"*» and «*ex:rex rdf:type ex:dog*».

<sup>10</sup> <http://quilter.dcc.fc.up.pt/antigraph>



## 4.1 Back end processing

The mapping of semantic graphs into antigraphs is performed in two stages by the back end. Firstly, a set of graph reductions is produced from the semantic graph triples. Secondly, the antigraph data is computed by processing these graph reductions.

A graph reduction instance aggregates edges of a single type, that is, the semantic graph obtained by considering only the edges of that type. It records the nodes that are sources and those that are targets, and computes their in and outdegrees. The links between these nodes are also recorded to compute aggregate measures on the reduction such as the number of cycles, depth and branching factor.

Graph reductions are computed by processing a stream of RDF triples. For each subject-predicate-object triple the reduction corresponding to its predicate the subject is selected and recorded as source and the object as target.

Each reduction corresponds to an antinode. Thus the second stage creates an antinode for each reduction found in the first stage, assigning it a weight computed as the percentage of edges in the graph. The top and bottom antinodes, with null weight, are also created. Then it iterates over the pairs of reductions to create antiedges.

The computation of antiedges' weights is more complex than that of antinodes, as it involves determining the intersection of the targets and source sets of nodes respectively of the source and target antinodes of each antiedge. Also, the contribution of each of these nodes depends both on their in(out)degrees on the reduction. The pairs of antinodes with nonnull weights create antiedges.

Antiedges connecting antinodes to top and bottom need also to be considered. These are created with the nodes that are not fully consumed to create antiedges among regular antinodes, following the same approach to compute weights. It should be noted that links between top and bottom are impossible.

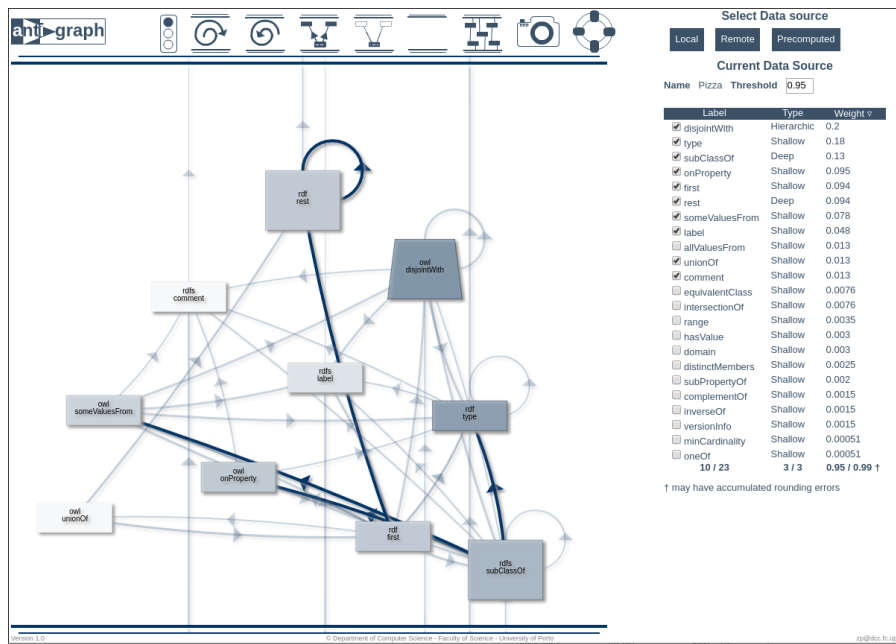
## 4.2 Diagram layout

Antigraphs serialized in JSON are sent to the front end where they are visualized as diagrams. The layout of these diagrams is computed using a force-directed algorithm [14]. Antinodes repel each other according to Coulomb's law as if they were electrically charged particles with the same signal. Antiedges bind them together as springs following Hooke's law.

The top and bottom antinodes, as well as the antiedges connecting them, are ignored in this process. The layout is performed in a rectangular area that acts as a boundary that confines regular antinodes. Top and bottom antinodes are positioned respectively at the top and bottom of this rectangle, and antiedges connecting them are plotted perpendicularly to them.

One of the advantages of a force directed algorithm is that it adjusts to changes, either of window dimensions or in the number of nodes. This enables the selection of antinodes, choosing which to display and which to hide, with the quick readjustment of the layout. When an antinode is hidden so are the antiedges that link to it.

Antigraphs with a large number of antinodes tend to have an even larger number of antiedges, which may difficult their visualization. In this case, the natural candidates to hide are those with smaller weight since they represent a smaller number of edges in the semantic graph. To simplify this kind of selection the antigraph browser provides a node weight threshold. If this threshold is provided then antinodes are sorted by weight and their accumulated weight is computed in this order. When this value exceeds the threshold the remaining antinodes are hidden, as well as the antiedges linking to them.



■ Figure 5 Antigraph browser.

### 4.3 User interface

Figure 5 depicts the user interface of the antigraph browser available online. The main part is the left central region where the diagram is displayed, following the approach described in the previous subsection. Above this area, there is a toolbar with tools for controlling the diagram layout. The smaller panel on the right contains a data source selector and displays the current data source features. The remainder of this subsection describes these panels in detail.

The antigraph browser has a number of features to control the diagram layout. These features are accessible through the icons on the header toolbar. To start with, the incremental layout can be toggled on and off using the traffic light icon, on the left of the toolbar. The icons to its left provide ways to show and hide antinodes, as well as the antiedges connecting them. The most relevant (with higher weight) hidden antinode is shown by pressing the outward spiral icon. Using this tool it is possible to gradually enlarge the diagram. The reverse tool, bound to the inward spiral icon, hides the least relevant shown antinode.

The following two icons operate on the currently selected antinode: to show all currently hidden antinodes connected to it, or to hide all antinodes connected to it. Antinodes are selected just by clicking on them. Clicking an antinode with the mouse's middle button also toggles a tool tip hovering the node. This tool tip displays the characteristics of the antinode, such as label, type and weight.

The hide all and show all tools allow the user to set the layout at the two extremes. These tools are respectively bound to the icons with an antigraph with no antinodes and the antigraph with several antinodes and antiedges. The header toolbar includes two other icons on its right side: the camera icon and the life saving icon. The latter opens a help window expanding the information in this paragraph.

The camera icon produces a *vector* image of the diagram presented in the browser. Using the normal browser features, it is possible to obtain a raster image of the diagram. However,

this kind of image is inadequate for publication since it has a fixed and typically low resolution. The camera icon activates a feature that produces an SVG file with the diagram, using the same layout algorithm described above. This conversion uses the SVGKit<sup>11</sup> package, that works well for graphic primitives (e.g. lines, rectangles, ellipses) but has some limitations regarding fonts and shadows. The vector images look slightly different from their raster counterparts, but have better quality when printed. The diagrams of the next section, as well as those of Subsection 3.2, were produced using this tool.

The antigraph browser presents a second panel next to the diagram. Depending on the width of the web browser's window, this panel may be placed either on the right side (as in Figure 5) or below the diagram. The panel contains a data source selector and displays the main features of the current data source.

The upper part of the side panel is used for selecting a semantic graph as data source for generating an antigraph. It provides three kinds of semantic graph sources: local, remote and precomputed.

Local sources include small examples for testing the basic features of antigraphs, and were presented in Subsection 3.2. The dialog box for the selection of local graphs presents the RDF triples that will be processed to produce the antigraph. These triples are in N-Triples format in an editable window. The user may modify, add or delete these triples, to better understand how these changes are reflected on the antigraph diagram.

The remote sources are RDF graphs available on the web in XML/RDF format. This dialog box presents each graph's URLs and a threshold – the weight above which antinodes are included in the diagram. The last entry of this dialog box allows the user to enter an URL to any RDF/XML file available on the web, and assign it an initial threshold. This threshold may be changed later on the current data source panel.

Both the local and remote data sources are processed on the fly by the server. The precomputed data sources provide access to larger semantic graphs that require long processing times and are already available on the client side. Most of these examples are analyzed in detail in the next section.

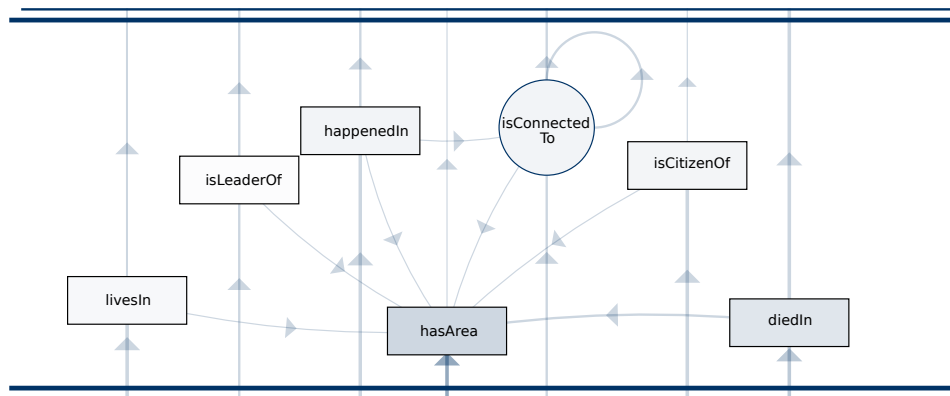
The current data source panel displays its name, threshold and a grid listing its antinodes. This grid lists all the antinodes in the antigraph, showing which are currently visible, their type and weight. By default, this information is ordered by descending antinode weight, but the user can change it. The user can also (de)select the visible antinodes, which immediately changes the diagram layout. Also, changes in the diagram resulting from the tools described above are also immediately reflected in this grid.

## 5 Validation

This section shows with concrete examples how antigraph diagrams emphasize the most relevant path patterns of a semantic graph. It also explains how the tools in the antigraph browser help the discovery of path patterns in large semantic graphs, by temporarily hiding some of their antinodes and the antiedges connecting them, thus producing meaningful diagrams with a reasonable small size.

---

<sup>11</sup><http://svgkit.sourceforge.net/>



■ Figure 6 Yago core – antinodes connecting to *hasArea*.

## 5.1 WordNet

Wordnet[8] 2.1, whose antigraph diagram is depicted in Figure 1, is a much larger graph than those presented in Subsection 3.2. However, this figure refers only to 95% of Wordnet 2.1 since the 5% least representative edges are omitted. By default, when this example is selected the threshold is set to 95%, but this value may be edited or removed in the corresponding field.

The WordNet 2.1 graph has 27 types of nodes and their corresponding antinodes would clutter this figure. This approach quickly produces a simple visualization by temporarily hiding the 2/3 least representative antinodes, i.e. edge types. It is important to point out that this is not specific of WordNet. All the semantic graphs tested with the antigraph browser have most of their paths concentrated in a fairly small number of edge types, hence this approach can be systematically used to improve the antigraph visualization.

This diagram immediately shows that the edges types that participate in most triples are from imported namespaces – *rdf:type* and *rdfs:label* – since the corresponding antinodes are darker. Two antinodes of the *wn20schema* namespace stand out from the pack for having links to several others, namely *hyponymOf* and *containsWordSense*, but the former participates in more “joins”, as evidenced by the darker antiedges.

WordNet is frequently used as a semantic proxy by path based semantic measures [10]. These measures rely on taxonomic relationships to identify a least common ancestor between two concept nodes and compute the shortest path between them. Taxonomic relationships are created using *partOf* (hierarchical) and *isA* relationships. For instance, the RDF and RDFS vocabularies provide the *rdf:type* and *rdfs:subclassOf* properties that can be used to create a taxonomic relationship between typed resources. However, in this version of WordNet *rdf:type* is available, but *rdfs:subclassOf* is missing.

The antigraph diagram in Figure 1 shows an alternative hierarchic relationship – *hyponymOf* – that complemented with other relationship can be used to create a taxonomic relationship. The *rdfs:label* relationship is connected by an antiedge with *hyponymOf*, hence they can be combined to create a taxonomic relationship on words.

Of course, this is not new knowledge. It is well known that WordNet can be used as a semantic proxy using *hyponymOf* and another property to create a taxonomic relationship. The point is that the antigraph diagram highlights the most promising candidates to create a taxonomic relationship. This should be useful to discover candidates for taxonomic relationships in even larger semantic graphs, such as DBpedia [4].

■ **Listing 1** SPARQL query to count leaders of geographic areas.

```
SELECT COUNT(*)
WHERE {
  ?p yago:isLeaderOf ?g.
  ?g yago:hasArea ?a.
}
```

■ **Listing 2** Counting places connected to where something happened.

```
SELECT COUNT(*)
WHERE {
  ?s yago:happenedIn ?g .
  ?g yago:isConnectedTo ?p .
}
```

## 5.2 Yago

Yago<sup>12</sup> [13] is a well known semantic knowledge base derived from several sources, such as DBpedia, WordNet, and GeoNames. It has over 10 million entities but for this study, only the core was used and labels were omitted. Still, this corresponds to over 20 million triples with 60 property types. Hence it produces an antigraph with that order and size 487. Even with a threshold of 80%, as it is by default on the antigraph browser, it is difficult to grasp.

The diagram in Figure 6 was obtained by selecting a single antinode, *hasArea*, the second most frequent edge type in this graph. Afterward, it was used the tool that unhides antinodes connected to the one currently selected. The point is to find property types related to concepts that have an area. Examples of such concepts would be cities, regions or countries. In a sense, *hasArea* can be seen as a defining property for a class of geographic concepts, although that is not explicit. The diagram shows that these geographic concepts are connected to other properties, such as *livesIn*, or *isLeaderOf*. That is, it is possible to retrieve information about who lives in or who is the leader of an concept that has an area. The SPARQL query in Listing 1 should produce a nonempty result set. In fact, it was checked on a Yago SPARQL endpoint<sup>13</sup> and the result is 5666.

Also, one can determine the area of entities where something happened, *happenedIn*, or that are connected to each other. The type of this last antinode is cyclic, meaning that it corresponds to a reflexive edge type. These two antinodes are the only that are directly connected without using *em hasArea*. Hence, it must be possible to obtain a non empty answer to the query “what places are connected to the place where something happened?”, using the query in Listing 2, and it actually returns 888 solutions.

Surprisingly, the graph also indicates that one should not expect results for the query “what places are connected to place that is lead by x” since these two antinodes are not connected. Running the SPARQL query in Listing 3 verifies that conclusion as the result is zero.

<sup>12</sup><https://www.mpi-inf.mpg.de/yago-naga/yago>

<sup>13</sup><https://linkeddata1.calcul.u-psud.fr/sparql>

■ **Listing 3** Are citizens connected to other places?

```
SELECT COUNT(*)
WHERE {
  ?s yago:isCitezenOf ?g .
  ?g yago:isConnectedTo ?p .
}
```

## 6 Conclusions and future work

Semantic graphs are hard to visualize due to a large number of typed nodes and edges. The antigraph approach to abstract semantic graphs maps edge information into antinodes and node information into antiedges. This reversal in the nature of the main constituents of a graph is the reason for the prefix “anti”. The abstraction mapping produces a smaller graph that is easier to visualize and highlights the patterns of paths in the original semantic graph.

Antinodes and antiedges are assigned with weights that reflect the relevance of the edges and nodes they represent, and that can be used for further abstractions. For instance, antinodes with small weights, corresponding to types of edges that seldom occur in the semantic graph, can be omitted to unclutter large antigraph diagrams.

The antigraph diagram is the proposed graphical syntax to represent antigraphs, and thus visualize the semantic graphs. This kind of diagrams uses different shapes to represent antinodes according to their types, and transparency to denote weights. The special antinodes top and bottom are represented as parallel lines respectively on the top and bottom of the diagram. In an antigraph, diagram paths are in general upwards, which facilitates their detection.

The web application for visualizing and interacting with antigraphs is also an important contribution of this research. It uses a force direct algorithm, which allows the incremental layout of the diagram after reposition or removal of antinodes. This application can use data from different sources: local data entered on the interface, remote data available on the web and precomputed data for a few preprocessed semantic graphs.

The proposed approach still faces the challenge of dealing with massive semantic graphs with millions of triples, such as those of Yago and DBpedia. The major problem is due to the computational complexity involving antiedge weights. However, there are approaches to curb this complexity that are currently being researched. After tackling this issue, the antigraph browser will be easier to evaluate with real users interested in discovering path patterns in large semantic graphs.

---

### References

- 1 James Abello, Frank Van Ham, and Neeraj Krishnan. ASK-GraphView: A large scale graph visualization system. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):669–676, 2006.
- 2 Daniel Archambault, Tamara Munzner, and David Auber. GrouseFlocks: Steerable exploration of graph hierarchy space. *IEEE Transactions on Visualization and Computer Graphics*, 14(4):900–913, 2008. doi:10.1109/TVCG.2008.34.
- 3 David Auber. *Tulip — A Huge Graph Visualization Framework*, pages 105–126. Springer, 2004. doi:10.1007/978-3-642-18638-7\_5.

- 4 Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives. DBpedia: A nucleus for a web of open data. In *The Semantic Web*, pages 722–735, 2007.
- 5 Mathieu Bastian, Sebastien Heymann, and Mathieu Jacomy. Gephi: An open source software for exploring and manipulating networks. In *8th International AAAI Conference on Weblogs and Social Media*, pages 361–362, 2009.
- 6 Fabio Benedetti, Sonia Bergamaschi, and Laura Po. A visual summary for linked open data sources. In *International Semantic Web Conference*, 2014.
- 7 Nikos Bikakis, John Liagouris, Maria Kromida, George Papastefanatos, and Timos Sellis. Towards scalable visual exploration of very large RDF graphs. In *The Semantic Web: ESWC 2015 Satellite Events*, pages 9–13. Springer International Publishing, 2015. doi:10.1007/978-3-319-25639-9\_2.
- 8 Christiane Fellbaum. *Wordnet: An electronic lexical database*. MIT Press, 1999.
- 9 Nicola Guarino, Daniel Oberle, and Steffan Staab. What is an ontology? In *Handbook on ontologies*, pages 1–17. Springer, 2009.
- 10 Sébastien Harispe, Sylvie Ranwez, Stefan Janaqi, and Jacky Montmain. Semantic similarity from natural language and ontology analysis. *Synthesis Lectures on Human Language Technologies*, 8(1):1–254, 2015.
- 11 Tuukka Hastrup, Richard Cyganiak, and Uldis Bojārs. Browsing linked data with Fenfire. In *Linked Data on the Web*, 2008.
- 12 Philipp Heim, Sebastian Hellmann, Jens Lehmann, Steffen Lohmann, and Timo Stegemann. RelFinder: Revealing relationships in RDF knowledge bases. In *Semantic Multimedia*, pages 182–187, 2009. doi:10.1007/978-3-642-10543-2\_21.
- 13 Johannes Hoffart, Fabian M. Suchanek, Klaus Berberich, and Gerhard Weikum. YAGO2: a spatially and temporally enhanced knowledge base from Wikipedia. In *Proceedings of the 23rd International Joint Conference on Artificial Intelligence*, pages 3161–3165, 2013.
- 14 Stephen G. Kobourov. Spring embedders and force directed graph drawing algorithms. *CoRR*, abs/1201.3011, 2012. URL: <http://arxiv.org/abs/1201.3011>.
- 15 Zhiyuan Lin, Nan Cao, Hanghang Tong, Fei Wang, U. Kang, and Duen Horng Polo Chau. Demonstrating interactive multi-resolution large graph exploration. In *Proceedings of the 2013 IEEE 13th International Conference on Data Mining Workshops*, pages 1097–1100, 2013. doi:10.1109/ICDMW.2013.124.
- 16 Kang Zhang, Haofen Wang, Thanh Tran, and Yong Yu. ZoomRDF: semantic fisheye zooming on RDF data. In *19th international conference on World Wide Web*, pages 1329–1332, 2010.
- 17 Michael Zinsmaier, Ulrik Brandes, Oliver Deussen, and Hendrik Strobelt. Interactive level-of-detail rendering of large graphs. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2486–2495, 2012. doi:10.1109/TVCG.2012.238.