Extending the Hierarchical Visualisation System (HVS) with a Radial Tree Visualisation and SKOS Import Functionality

Matthias Fuchs

Extending the Hierarchical Visualisation System (HVS) with a Radial Tree Visualisation and SKOS Import Functionality

Bachelor's Thesis

at

Graz University of Technology

submitted by

Mag. Matthias Fuchs

Institute for Information Systems and Computer Media (IICM),
Graz University of Technology
A-8010 Graz, Austria

24 Apr 2015

Advisor: Ao.Univ.-Prof. Dr. Keith Andrews

Erweiterung des hierarchischen Visualisierungssystems (HVS) mit einer radialen Baumvisualisierung und SKOS-Import-Funktionalität

Bakkalaureatsarbeit

an der

Technischen Universität Graz

vorgelegt von

Mag. Matthias Fuchs

Institut für Informationssysteme und Computer Medien (IICM),
Technische Universität Graz
A-8010 Graz

24. April 2015

Diese Arbeit ist in englischer Sprache verfasst.

Begutachter: Ao.Univ.-Prof. Dr. Keith Andrews

Abstract

Information visualisation provides numerous approaches to visually represent hierarchies and interact with them, thereby helping users gain an insight into the extent and structure of the hierarchies. There are also numerous data formats used to store hierarchies, including a number of open, standardized formats.

This thesis focuses on both aspects: representing hierarchies visually as well as in an exchangeable data format. The radial tree browser provides various radial tree layouts for hierarchy visualisation and also supports interactivity. It is implemented by extending the Hierarchical Visualisation System (HVS), which is a portable framework for tree visualisation written in Java. HVS is extended with a parser for the Simple Knowledge Organization System (SKOS), making both the structure and assigned semantic information of a hierarchical knowledge organisation system realised with SKOS available to HVS and its visualisations.

Kurzfassung

Die Informationsvisualisierung stellt reichlich Ansätze zur Visualisierung und auch Interaktion mit Hierarchien bereit, die Anwender beim Erkennen von Struktur und Umfang von Hierarchien unterstützen. Darüber hinaus gibt es viele Datenformate Hierarchien zu repräsentieren, von denen vor allem austauschbare Datenformate von Interesse sind – schließlich fördern diesen den Diskurs.

Diese Bakkalaureatsarbeit beschäftigt sich mit beiden Aspekten: Das visuelle Darstellen von Hierarchien sowie deren Repräsentation in einem austauschbaren Datenformat. Dazu stellt der radiale Baum-Browser verschiedene radiale Bäume bereit und ermöglicht auch einen interaktiven Umgang mit diesen. Für die Implementierung wird das in Java geschriebene hierarchischen Visualisierungssystem (HVS) erweitert. HVS wird auch erweitert um mit dem *Simple Knowledge Organization System* (SKOS) erstellte Daten einlesen zu können, um diese allen HVS-Visualisierungen verfügbar zu machen.

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Place	Date	Signature

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Ort	Datum	Unterschrift

Contents

List of Figures

List of Tables

List of Listings

Acknowledgements

I especially wish to thank my advisor, Keith Andrews, who provided both constant support and advice over the course of many months.

Matthias Fuchs
Graz, Austria, April 2015

Credits

I would like to thank Keith Andrews for providing a recent version of his LaTeX skeleton [Andrews, 2014] with which this thesis was written.

Figure 4.1 on page 16 was extracted from Otlet [1934, page 420], scanned in by the Universiteitsbibliotheek Gent, and is used under the terms of the Public Domain Mark 1.0 license [Creative Commons, 2010].

Chapter 1

Introduction

This thesis describes the enhancement of the Hierarchical Visualization System (HVS) [Andrews and Putz, 2005] with radial tree visualisations and import functionality for hierarchies stored in the Simple Knowledge Organization System (SKOS) [Miles and Bechhofer, 2009] format. The following chapter provides an introduction to information visualisation. Chapter 3 focuses on hierarchies and their visualisation, describing different techniques.

Chapter 4 discusses radial trees in more detail and also highlights the differences and trade-offs between the presented radial tree layouts. A very simple algorithm is improved and extended over the course of the chapter. The implementation of radial tree layouts in HVS, including possible user interactions, is discussed in Chapter 5. Chapter 6 describes the structure of SKOS, including the SKOS specific vocabulary. The implementation of the SKOS import functionality for HVS is outlined in Chapter 7. In addition, the considerations for choosing the libraries upon which to base SKOS support on are also reviewed.

Chapter 8 lists some of the many changes to HVS that not only simplified the code base, but also lead to more consistent and intuitive behaviour of HVS. Appendix A provides a user guide for the radial tree browser, the SKOS functionality, and for the other introduced changes.

Chapter 2

Information Visualisation

Spence [2007, page 5–6] describes information visualisation as presenting data in a concise way that makes it easy for humans to gain insight and to derive information from the presented data. Thus, the data is transformed a way that fits human cognitive capabilities.

The presentation method of data heavily influences the mental model humans create. For instance, by choosing a certain scale of the axes of a diagram it is possible to influence and even manipulate the perceived insight gained [Ward, Grinstein, and Keim, 2010, pages 3–4]: Scattered dots might appear well distributed on both the x- and y-axis or the very same data might appear spatially close like in a cluster, or vertically or horizontally distributed. As a result, care must be taken to ensure that data is presented clearly.

Whenever visualising data, it is useful to exploit the characteristics of human visual perception. As Few [2013, Chapter 5] points out, humans are able to instantly, without thinking attentively and thus pre-attentively, recognise certain stand-out features which can be grouped into colour, form, spatial position, and motion. When there are multiple dots, humans easily recognise a dot with a different colour, or a dot with a different size. Moreover, two objects are automatically recognised as grouped, when they are close together or when they are connected with a line even if they are far apart. Utilising these and more techniques enables the representation of complex data a clear and easy understandable way.

Depending on the underlying data there are different types of information. Visualisations are often tailored for specific types of information [Andrews, 2015, page 14], for example:

- Linear: text, chronological, ...
- Hierarchies: tree structures.
- Networks: graphs of nodes and links.
- Multidimensional: tables, spreadsheets, metadata attributes.

Of these, hierarchies and their visualisations will be investigated in more detail in the following chapters.

Chapter 3

Hierarchy Visualisation

Hierarchies are everywhere around us, ranging from hierarchies in organisations like governments, corporations or schools to more abstract hierarchies like file systems on computers. Hierarchies have a huge advantage of breaking down vast amounts of data into smaller manageable pieces. A general is not concerned with a soldiers' everyday life, which is the soldiers' direct superior's job, and instead focuses on operational and strategic concerns. At the same time, users are not generally concerned with all the files on their hard drives at once. Instead, they can focus on just a small part of the hierarchy they are interested in. Sometimes, the whole structure of a hierarchy might be of interest. A useful visualisation of a hierarchy depends first and foremost on the user's task, as do the possible interactions with the visualisation.

Visualising hierarchies can help in understanding them and even in gaining insight that did not exist before. Since any hierarchy can be visualised by a tree, these terms are often used interchangeably [Ward, Grinstein, and Keim, 2010, page 272] which is also done here to some degree. The distinction made here is that the term "tree" will be used directly in connection with a visualisation, most times a simple node-link tree visualisation, while "hierarchy" will be used to describe hierarchically structured data.

The following categorisations are often used for tree visualisations:

- Node-link (explicit) trees.
- Space-filling (implicit) trees.

Node-link trees refer to tree visualisations where the nodes are visually connected by links indicating parent-child relationships. Such trees are also called explicit trees. In contrast, space-filling trees, also called implicit trees, do not use explicit visual links, but instead segment space into areas which represent nodes, the location of these areas illustrates the relationship between the nodes.

The following sections give an overview of hierarchy visualisations: providing historical context, giving examples, and emphasising requirements, limitations, and possibilities of hierarchy visualisations. For a more detailed overview of different hierarchical visualisations, readers can consult treevis.net [Schulz, 2015].

Hierarchy visualisation is a topic with a long history. There is an abundance of papers on hierarchy visualisation and, as a result, some visualisation approaches haven been (re)invented multiple times by different people [Schulz, 2011, page 11]. To overcome the issue of not finding relevant sources, treevis.net [Schulz, 2011] was created. treevis.net makes it easy to search for papers on hierarchy visualisation. Users can constrain their search by providing categories of trees they are interested in. This allows users, for example, to easily find relevant papers on three-dimensional, radial tree visualisations containing a given keyword.

3.1 History of Tree Visualisations

This section gives a short historical overview of using trees to visualise hierarchies. It is mostly based on Lima [2014, pages 15–47]. Lima [2014] focuses on the history of using trees for hierarchy visualisation and presents different forms of tree visualisation with illustrations of both historical and recent examples.

Trees (the botanical form) have fascinated humans for centuries. Trees not only provide resources which can be used in a multitude of ways like for tools, energy, building, food and many more applications, but also symbolise strength and stability. Some trees can become older than 1000 years. Broad-leaved trees are often used to illustrate the seasons, moreover their transformations are also often used to symbolise the cycle of life with its different phases.

Trees appear as symbols in many cultures as well as religions around the globe. For example, Christians have the tree of life and the tree of knowledge, while ancient Egyptians worshipped specific trees. The rootedness of trees in our culture also influenced language. Tree-related terms like "root" or "branch" are common in most languages beyond their botanical meaning, as was demonstrated in the previous sentence with "rootedness". Moreover, the expression "to get to the root of a problem" exists at least in Germanic and Latin-based languages.

Porphyrian trees are recognised as the foundation of tree visualisations. Porphyry, a Greek philosopher, resorted to text to describe Aristotle's order of nature, yet tree visualisations were later created based on the descriptions and ideas of Porphyry's work. From then on, trees were used in classifications and gained popularity in the Middle Ages. Very common uses were genealogical trees, trees categorising laws and decrees, and especially trees used in exegesis. These visualisations made abstract concepts easier to understand and gave an overview, where details were often given in the accompanying text. Different features of trees were used. In some illustrations, the information is presented in fruits and leaves, while others put information in roots, branches, and the trunk. Furthermore, God and Jesus were often placed in the treetop in religious works. One of the oldest tree visualisations are bishop Isidore Seville's Consanguinity Trees of the seventh century [Meirelles, 2013, page 24]. Of antic family trees used as decoration in the buildings of wealthy Roman citizens, only textual descriptions remained [Kruja et al., 2002, page 273].

Trees also gained popularity in biological works to visualise relations between species, for example Augustin Augier's visualisation of plant relationships in 1801 [Pietsch, 2013, page 32], which cumulated in Charles Darwin's evolution theory, where a tree was the centrepiece to communicate the idea of common descent, evolutionary selection, and ongoing evolution. Interesting in this regard is that Darwin used a very abstract tree, containing annotated connected lines with no obvious resemblance to botanical trees [Lima, 2014, pages 38–40].

When a trend to abstract away from botanical trees to more stylised versions started is not clear according to Lima [2014, page 40]. Some illustrations from the Middle Ages, like the biblical genealogy by Stephanus Garsia Placidus around 1060 [Lima, 2014, page 112], were already very abstract. Essential to these abstractions is that no information is lost. In contrast, it can be argued that very figurative illustrations distract readers by adding visual noise.

Starting in the 19th Century, further forms of node-link trees like radial trees, which are the focus of this work, were devised. The advent of personal computers with capable displays and vast amounts of data resulted in the creation of many new tree visualisations at the end of the 20th Century. Some of these are discussed in the following sections. From then on, interaction with visualisations gained importance and is now a central part when visualising both small and large hierarchies on computer displays. Interaction can be as simple as linking to the underlying data or opening a file in the associated program, or as complex as navigating deep hierarchies, applying different filters, and dynamically modifying the hierarchy.

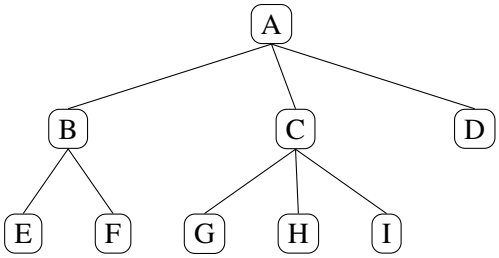


Figure 3.1: A simple hierarchy visualised as a basic tree with the root at the top. [Diagram generated by the author of this thesis using the TikZ language [Tantau and Feuersaenger, 2014; Chiang, 2012] for TEX.]

3.2 Node-Link Tree Visualisations

Tree visualisations exist in different variants, there are both two-dimensional and three-dimensional tree visualisations, even combinations of two-dimensional and three-dimensional tree visualisations exist, like the one described by Nguyen, Ho, and Shimodaira [2000]. Furthermore, trees can be drawn in different geometries like Euclidean geometry and hyperbolic geometry.

Figure 3.1 gives an example of a node-link tree visualisation, which is sometimes called a *basic tree*, *vertical tree* [Lima, 2014, page 79], or *traditional tree* [Burch et al., 2011]. In such a tree, the root node, here node A, is placed on top while its children are placed below. Every child node is connected to its parent with a link. A variation of this layout places the root node at the bottom with the children above. In contrast, horizontal trees [Lima, 2014, page 97] place the nodes from left to right, or right to left, thus arranging them horizontally.

More general graphs (node-link structures) can also be presented with tree visualisations. Unlike trees though, graphs such as peer-to-peer networks have no root and thus do not represent a hierarchy. Graphs can be visualised as trees by choosing any node as root. Sometimes the node with the lowest average distance to all the other nodes is chosen [Eades, 1992]. Skiena [2008, page 517] points out that using trees to visualise graphs might mislead by suggesting a hierarchy which is not there. Nonetheless, hierarchical methods are often used to visualise graphs and as a result related papers like Yee et al. [2001] and the described techniques are of interest here.

3.2.1 2D Node-Link Tree Visualisations

An example for a basic tree was given in Figure 3.1, the same hierarchy can also be visualised with the commonly used Walker [Walker, 1990] algorithm like in Figure 3.2, which leads to more pleasing results. A central idea of the Walker algorithm is to first layout subtrees independently, separating nodes with a fixed distance d. When two subtrees are combined with a common ancestor, the right subtree is placed on the right of the left subtree, ensuring that each node of the left subtree has at least the separation d to the nodes of the right subtree. Parent nodes are always centred above their children. Using the algorithm described above it is possible that smaller subtrees are clustered between their siblings with larger subtrees, which the Walker algorithm counteracts by distributing these smaller subtrees equally.

In Figure 3.3 the hierarchy of Figure 3.1 is visualised as an orthogonal tree, using the Walker algorithm. Every edge in an orthogonal tree drawing is orthogonal. A very common use of orthogonal trees is to display the directory structure of a computer file system, using visualisations sometimes called outliners [Andrews, 2015, pages 33–34]. Outliners are like horizontal orthogonal trees, which allow collapsing and expanding of nodes. Many of these visualisations like the outliner in Dolphin [Penz, 2012] in Figure 3.4 also display files alongside directories.

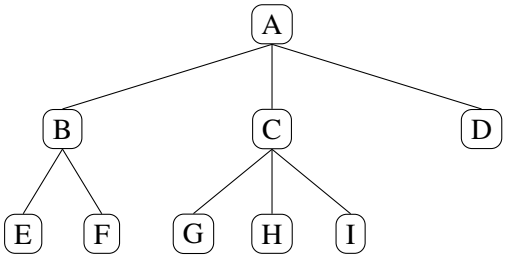


Figure 3.2: A hierarchy visualised with the Walker [Walker, 1990] algorithm. [Diagram generated by the author of this thesis using the TikZ language [Tantau and Feuersaenger, 2014; Chiang, 2012] for TEX.]

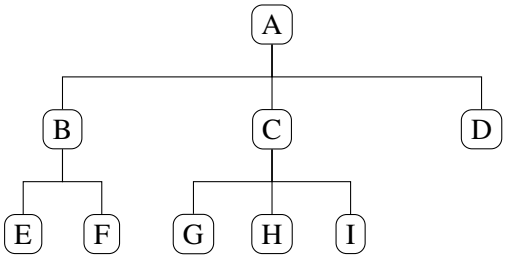


Figure 3.3: An orthogonal tree drawing uses only orthogonal edges to connect the nodes. [Diagram generated by the author of this thesis using the TikZ language [Tantau and Feuersaenger, 2014; Chiang, 2012] for TEX.]

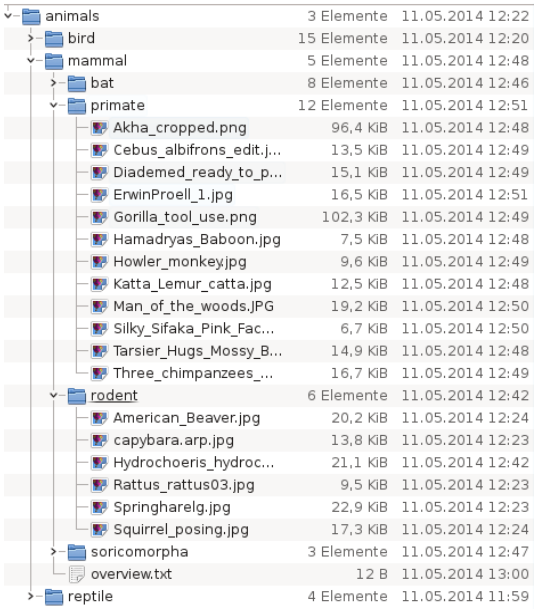


Figure 3.4: The file system hierarchy in Dolphin [Penz, 2012] is represented using an outliner visualisation. [Screenshot taken by the author of this thesis.]

3.2.2 3D Node-Link Tree Visualisations

Cone trees are a well known example of a 3D technique to display hierarchies. In a cone tree, each inner node is at the top of a cone, while its children are located at the base of that cone. Cone trees tend to have problems visualising large hierarchies with more than 1000 nodes. Often, nodes are occluded by others and there is much visual clutter. Carriere and Kazman [1995] tried to remedy many of the issues related to cone trees by using the following principles:

- Reducing visual clutter.
- Using animations to help the user keep the mental model of the hierarchy during navigating.
- Using shape to encode information.
- Using colour to encode information.
- Support interactivity like drag and drop support.
- Filtering techniques.
- Automatic coalescing of distant nodes into a single structure.

Many of these principles are related to utilising the pre-attentive capabilities of humans summarised by Few [2013, Chapter 5].

3.3 Space-Filling Tree Visualisations

In contrast to node-link trees, where a hierarchy is visualised by connected nodes of some form, space-filling techniques use filled areas to display trees. Very popular algorithms are based on rectangular partition (so-called treemaps), and radial partition of space into areas. These algorithms have in common that the available space is used very efficiently and split into areas proportional to some metric in the data.

An advantage of such techniques is the ease of visualising contextual information directly in the structure. For example, the size of a node area could depend on the total number of nodes below the node. When looking at a file system, the size could depend on the overall file and directory size in bytes. An area representing a directory would receive more space if the files and directories inside use more hard disk space. Similarly, a huge file like a video would utilise much more of the available space [Ward, Grinstein, and Keim, 2010, Chapter 8]. This makes space-filling visualisations very useful when trying to find the files and directories which occupy most hard disk space. For all common platforms, there are several programs available for the purpose of examining hard disk usage.

3.3.1 Treemaps

Treemaps represent a hierarchy using nested rectangles. The outermost rectangle represents the root node, it receives all available space. The root rectangle is subdivided into as many rectangles as it has direct children. How the rectangle is subdivided depends on the used algorithm. Child rectangles are in turn subdivided depending on their children. As a result, rectangles denote nodes and include all their children. The size of the rectangles is flexible. It could depend, for instance, on the number of successor nodes or on some other metric of attribute of the data.

KDirStat [Hundhammer, 2006] uses a treemap to examine disk usage of a selected directory or hard disk. The size of the rectangles depends on the space the files or directories occupy on the disk. Figure 3.5 displays the author's home directory. The window is organised into two sections: an outliner on

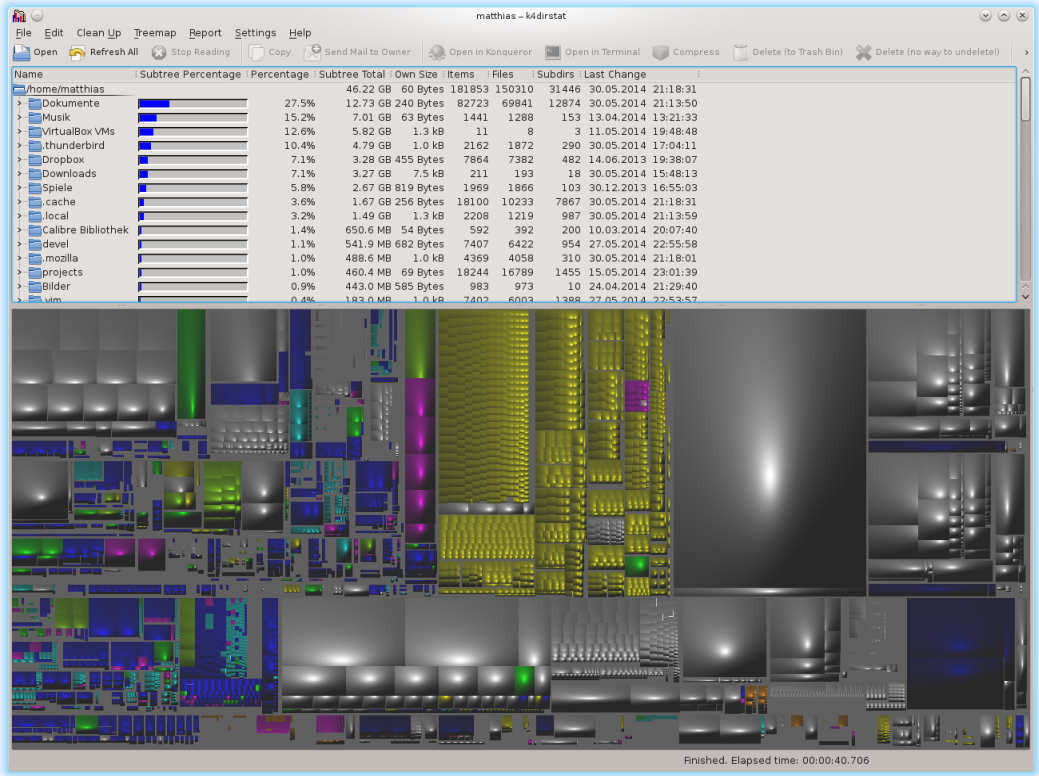


Figure 3.5: Representation of the author's home directory using the treemap visualisation of KDirStat [Hundhammer, 2006]. The area of the rectangles (files) represents the relative hard disk space the files occupy, colour denotes type. [Screenshot taken by the author of this thesis.]

top visualising the hierarchy sorted by disk usage and a treemap below. Both visualisations combined support finding and recognising very large files and the directories which contain them.

3.3.2 Radial Visualisations

Radial space-filling visualisations use concentric rings to display information. A popular radial visualisation is Sunburst [Stasko and Zhang, 2000]. In Sunburst, the root node is placed in the centre of the screen as a filled circle or disc. Then for each level in the tree, there is a ring around the inner disc. These rings are split into segments for leaves and inner-nodes. Defining the segment size works similarly to treemaps, thus is often based on child count or other criteria like file size.

One program using Sunburst is Filelight [KDEWiki, 2011], which is another program examining hard disk usage. Figure 3.6 shows the author's home directory visualised with Filelight. Some techniques used in Filelight to improve usability include:

- Utilising different colours to make the structure easier comprehensible.
- Hiding small files by default.
- Not displaying the whole hierarchy at once.
- Only showing labels for the children of the currently focused node, and then only for subsets of significant size.

Restricting the depth of displayed levels is especially useful for this use case, since Filelight is used to find places on the hard disk which occupy most space. Displaying the complete hierarchy at once is not necessary and therefore is omitted for better clarity.

3.4 Tree Visualisation Requirements and Limitations

In general, there are many requirements for tree visualisation algorithms. A tree should not only be visually pleasing, but also make it easy to understand the underlying data and to draw conclusions. The media used to display trees adds further requirements and limitations. When trees are displayed on a computer screen, interactivity is often a requirement to not only navigate the tree but also to counter limitations like the limited available screen space [Nguyen, Ho, and Shimodaira, 2000].

Technical limitations aside, humans also introduce limitations: for example, visual clutter distracts humans from the data and makes comprehension harder. As observed by Few [2013, pages 1–2], inapt systems for humans are still created on a large scale, ignoring the limitations and potentials of human abilities.

Many different criteria have been proposed to describe what a tree should look like to be visually pleasing and useful. The following criteria based on Nguyen, Ho, and Shimodaira [2000], Eades [1992], Herman, Melançon, and Marshall [2000], and Skiena [2008] describe aspects which a tree visualisation algorithm or computer program utilising a tree algorithm should fulfil. While the focus is on two-dimensional tree visualisations, many criteria also apply to other hierarchical visualisations:

- Comprehension: It is easy to understand the hierarchy.
- Efficiency: Available space is used economically.
- Interactivity:
  - Real time navigation through the tree.
  - Visual feedback on events like node removal or jumping to a distant node.

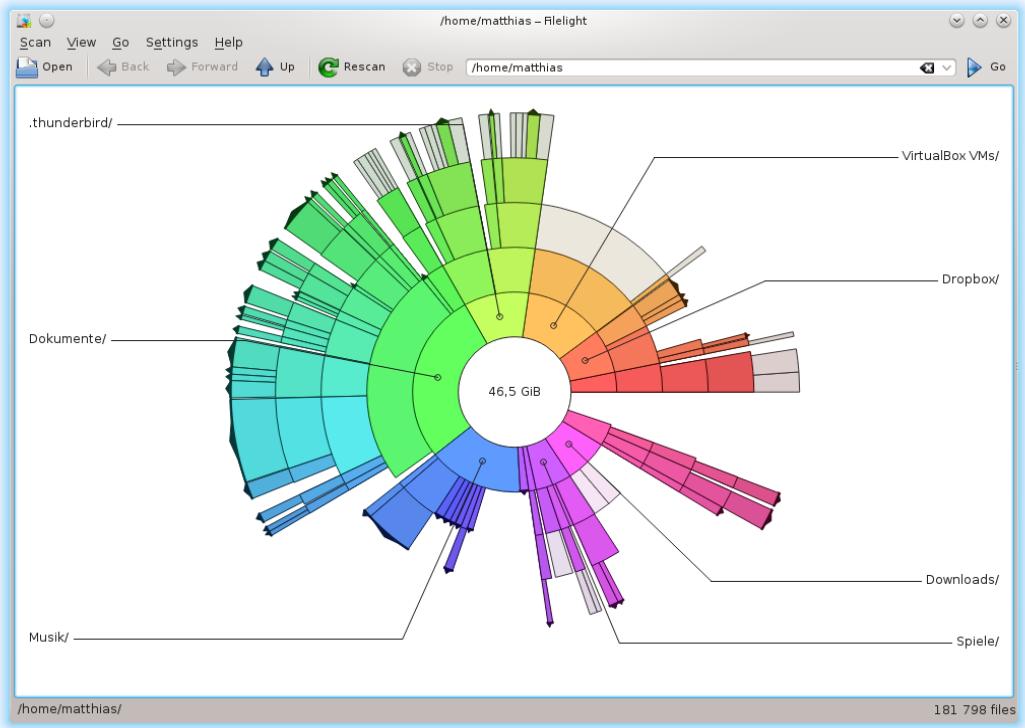


Figure 3.6: Representation of the author's home directory using the Sunburst visualisation of Filelight [KDEWiki, 2011] on Linux. The area of a slice (wedge) represents the relative hard disk space directories and all their children occupy. [Screenshot taken by the author of this thesis.]

- Aesthetics:
  - Avoiding edge crossings as these can make comprehension harder.
  - Avoiding long edges.
  - Edges not being too close as they become indistinguishable.
  - Uniform edge lengths.
- Predictability: Similar hierarchies should look similar.

These criteria are not to be interpreted as a fixed set of rules one has to abide by, as some of them are contradictory. Due to the lack of extensive usability studies, the order of importance of these rules is also unclear. Herman, Melançon, and Marshall [2000] suggest that rare edge crossings do not pose a problem. In some cases, for instance, when a tree is transformed dynamically, edge crossings are very hard to avoid, thus relaxing the requirements makes it easier to create implementations. Furthermore, fulfilling all these criteria becomes harder, the larger the hierarchy becomes. When space is used very efficiently, it is likely that comprehension of the structure is hard for huge hierarchies, since edges will be very close to each other. Since nodes tend to become indistinguishable in very large hierarchies, interactivity might become a problem. Herman, Melançon, and Marshall [2000] mention interactive techniques like zooming/panning, focus+context, incrementally exploring hierarchies, and search and filter, as possibilities to ameliorate these issues.

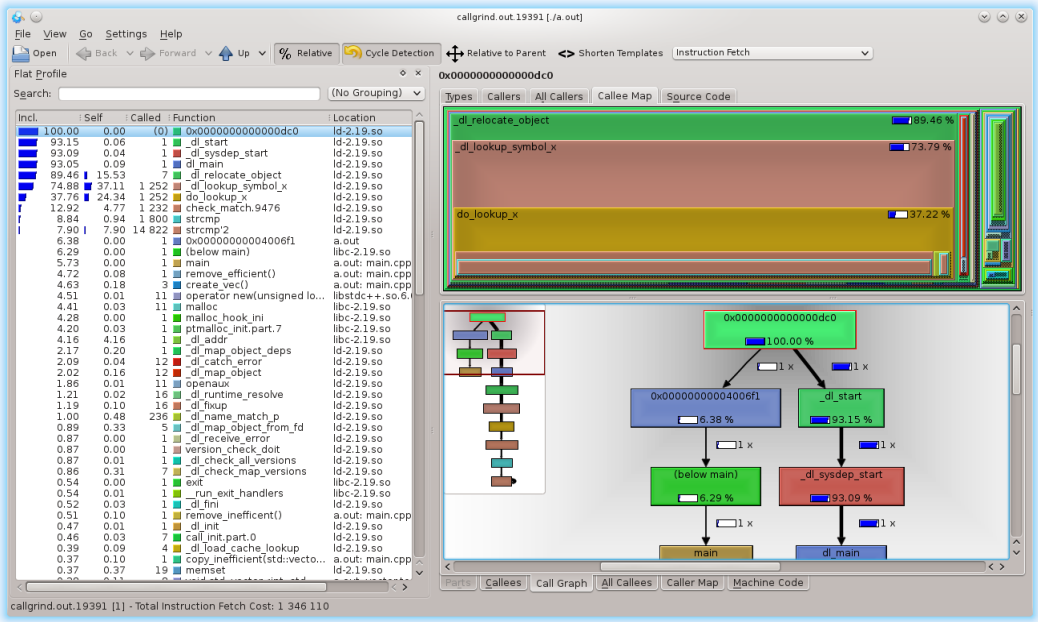


Figure 3.7: Multiple different visualisations are used in KCachegrind [Weidendorfer, 2013] to help software developers during profiling. [Screenshot taken by the author of this thesis.]

3.5 Coordinated Multiple Views

So far, different visualisations were presented more or less independently of each other. Yet it can be useful to combine different (but synchronised) visualisations of the same hierarchy to gain better insight. For example, file browsers like Windows Explorer often use an outliner for a top-level view, while the contents of a directory can be investigated in more detail using a different view alongside the outliner. KDirStat shown in Figure 3.5 also uses two different visualisations of the same hierarchy.

Another example is KCachegrind [Weidendorfer, 2013]. KCachegrind is a program to visualise profiling data collected with Callgrind [Weidendorfer, Kowarschik, and Trinitis, 2004], which itself is part of the Valgrind framework [Nethercote and Seward, 2007; Valgrind, 2013]. KCachegrind uses a multitude of different visualisations to help software developers during profiling tasks. When profiling, areas in the program which take a long time to execute are of special interest, since improvements there are most efficient. The task of finding these so called “bottlenecks” is similar to the already described file system examination use case, and as a result similar visualisation techniques are used. For example, in Figure 3.7 a tree map shows the used CPU cycles of function calls originating from a selected function. A basic tree shows the functions calling the selected function and the functions it calls in turn. An additional list gives an overview of all called functions. Strictly speaking, the underlying data originates from a graph and not a tree, since a function can be called from multiple sources and can call itself, but in this case a simple hierarchical visualisation helps to understand the underlying problem.

As a result, visualisation techniques should not be analysed in isolation, but instead possible combinations should be investigated which might increase the applicability of the tool. This is one target of HVS, which provides different synchronised hierarchical visualisations of the same hierarchy next to each other.

Chapter 4

Node-Link Radial Tree Visualisations

Radial trees display the root node of a hierarchy in the centre of a circle or a disc, surrounded by its descendants, each positioned on concentric circles depending on their depth. Direct children of the root are placed on the closest circle surrounding it, grandchildren on the circle thereafter, and so forth. The concept of radial trees has existed for more than 100 years. For example, Meirelles [2013, page 27] describes a radial tree used by Heinrich Gustav Adolf Engler in 1881 to visualise relations between plants. Figure 4.1 shows a radial tree visualisation of the organisation of the world, created by Paul Otlet in 1934 [Otlet, 1934, page 420].

In Figure 4.2, the hierarchy presented previously in Figure 3.1 on page 7 is visualised using a radial layout. The hierarchy is harder to comprehend, since it is not immediately obvious which node represents the root. That could be fixed using a larger radius for the circles, although then more space would be required. Another solution would be to explicitly display the concentric circles. In their usability study of different tree algorithms, Burch et al. [2011] noted that many test users had similar problems identifying the root of the tree. As a result, all following radial tree representations will also show the concentric circles, albeit in a lighter colour. This change can be seen in Figure 4.3 which is identical to Figure 4.2 except that concentric circles are shown. The hierarchy is more recognisable now thanks to the "Gestalt principle" of connection [Few, 2013, page 91]. Human perception assigns a lower priority to the circle connections than those of the edges since they use a lower colour intensity [Few, 2013, pages 82–83].

There are multiple ways to categorise radial tree visualisation algorithms. The author decided on the following categorisation:

- Fixed-Wedge Layouts
- Flexible-Wedge Layouts

Both categories have in common that radial space is assigned to nodes according to criteria like the number of direct children or the total number of descendants of a node. To illustrate the various algorithms, drawings were created with Python [Python, 2014] with pseudocode of the form shown in Listing 4.1. `draw` is a function taking two parameters `node1` and `node2`, which draws `node1` at its position and connects it with `node2` via an edge. Nodes contain their position in polar coordinates. If the two nodes are equal, only `node1` is drawn without an edge. `drawChildNodes` is the function containing the layout algorithm. It does not take care of the root node, which is manually handled at the bottom of the listing. Of course, the root node is positioned at the centre of all circles. Assume from now on, that every following algorithm listing contains the draw function as well as the initial `draw` and `drawChildNodes` calls, since these are omitted for more compact presentation. Moreover, the pseudocode does not focus on performance, but rather on simplicity, which may result in the same calculation being performed multiple times.

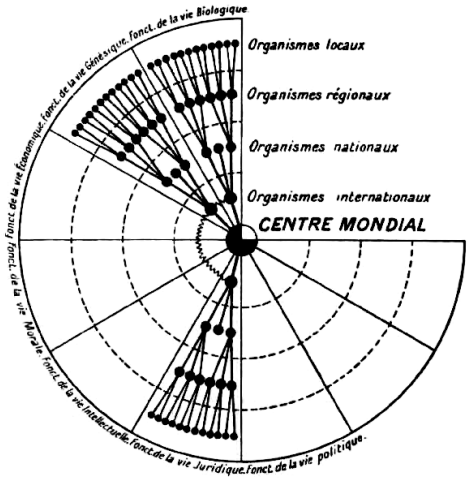


Figure 4.1: Paul Otlet's radial visualisation of the organisation of the world [Otlet, 1934, page 420]. [Image extracted from Otlet [1934, page 420], scanned in by the Universiteitsbibliotheek Gent, and used under the terms of the Public Domain Mark 1.0 license [Creative Commons, 2010].]

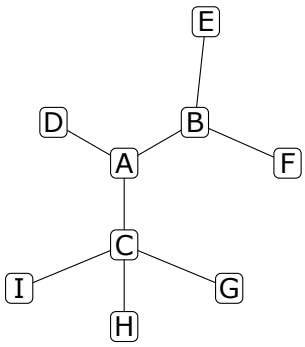


Figure 4.2: A radial tree of the same hierarchy shown in Figure 3.1 on page 7. The structure of the hierarchy is harder to recognise in the radial tree because it is not immediately obvious which node is the root. [Diagram generated by the author of this thesis using the Python programming language [Python, 2014].]

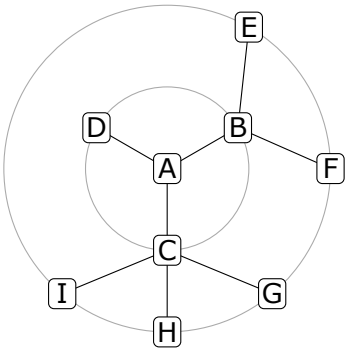


Figure 4.3: The same drawing as in Figure 4.2, but this time displaying concentric circles to make the hierarchy clearer. [Diagram generated by the author of this thesis using the Python programming language [Python, 2014].]

```
function drawChildNodes(parentNode, leftLimit, arcLength)
...
end

function draw(node1, node2)
...
end

draw(root, root)
drawChildNodes(root, 0, 2*PI)
```

Listing 4.1: The pseudocode used to draw radial trees. `drawChildNodes` contains the actual algorithm.

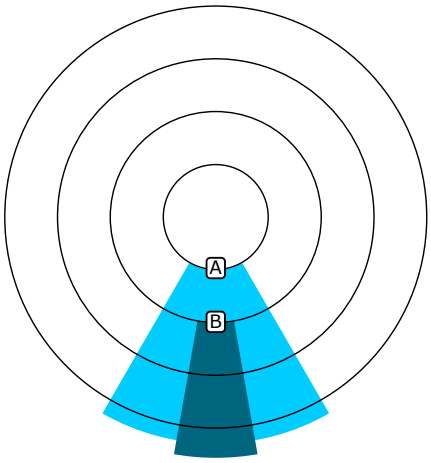


Figure 4.4: In fixed-wedge layouts the dark blue wedge of child B cannot be placed outside of its parent's light blue wedge. The same holds true for any parent child relation. [Diagram drawn by the author of this thesis using Inkscape [Inkscape, 2014].]

Multiple radial tree algorithms are investigated in the following sections, ranging from simple to more sophisticated ones. Specifically crafted hierarchies are used to highlight both the advantages and weaknesses of an algorithm.

4.1 Fixed-Wedge Layouts

In fixed-wedge layouts, every node is assigned a wedge of the circle where its children will be placed. This wedge is constrained by the parent's wedge. Thus no descendant node can be placed outside of the wedge of any predecessor node. Battista et al. [1999, pages 52–53] call this an annulus wedge. Figure 4.4 illustrates this concept: the light blue wedge is an annulus wedge for node A. The node and all its children must be contained inside this annulus wedge,as is illustrated by the dark blue wedge of node B. The location and size of the child wedge depends on the particular algorithm. If there is only one child, the child wedge might match the parent wedge.

4.1.1 Centring Nodes in the Annulus Wedge

Listing 4.2 describes a very basic algorithm for fixed-wedge layout, where each sibling node receives the same amount of space. Nodes are placed on the left of their annulus wedge, which can lead to irritating tree layouts depending on the hierarchy. While Figure 4.5a has a pleasing layout, Figure 4.5b looks irritating. It is important to note that the same algorithm was used for both figures and that Figure 4.5b

```
function drawChildNodes(parentNode, leftLimit, arcLength)
  for child in parentNode.children
    childArcLength = arcLength / parentNode.numChildren
    child.azimuth = leftLimit

    draw(child, parentNode)
    drawChildNodes(child, leftLimit, childArcLength)

    leftLimit = leftLimit + childArcLength
  end
end
```

Listing 4.2: Pseudocode for a very basic radial tree algorithm where each sibling receives the same amount of space and nodes are placed on the left of their annulus wedge.

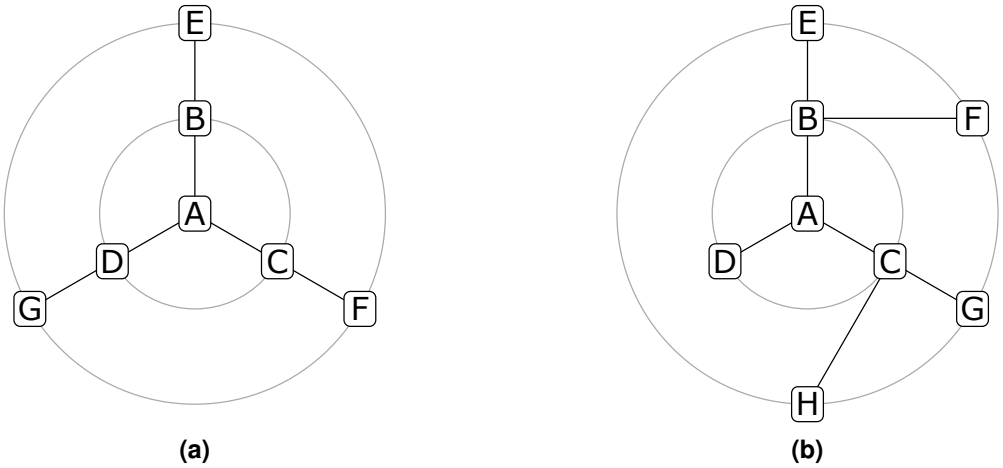


Figure 4.5: Both figures were created with the algorithm described in Listing 4.2, which places nodes on the left side of their annulus wedge. The negative effects of such placement strategy are visible for many hierarchies, like the one in (b). Only very simple hierarchies like in (a) do not have these negative effects. [Diagrams generated by the author of this thesis using the Python programming language [Python, 2014].]

simply highlights a weakness of the algorithm which was not visible in Figure 4.5a due to the particular hierarchy structure. This example also highlights a significant characteristic of all algorithms described here: the results of an algorithm are highly dependant on the particular hierarchy. That is why there are no perfect algorithms, there are always trade-offs to consider. In this case, more pleasing results can be achieved by centring nodes in their annulus wedge, which is done in Listing 4.3 and shown in Figure 4.6.

4.1.2 Restricting the Annulus Wedge

When considering the size of the annulus wedge, it is also important to decide whether it should be restricted or not. If the wedge is not limited, it is possible for edges to escape the ring the nodes are on, as discussed by Battista et al. [1999, pages 52–53]. Figure 4.7b shows an example what escaped edges can look like, using the algorithm described in Listing 4.3. Node B has an annulus wedge covering the whole circle, thus all children of B use the given space, in this case equally. The reason for the annulus wedge covering the whole circle is that B has no siblings which could restrict the wedge.

Book and Keshary [2001] describe a method to avoid such issues by restricting the annulus wedge to a maximum size, which is illustrated in Figure 4.8. The maximum annulus wedge has the arc s, which is

```
function drawChildNodes(parentNode, leftLimit, arcLength)
  for child in parentNode.children
    childArcLength = arcLength / parentNode.numChildren
    child.azimuth = leftLimit + childArcLength / 2

    draw(child, parentNode)
    drawChildNodes(child, leftLimit, childArcLength)

    leftLimit = leftLimit + childArcLength
  end
end
```

Listing 4.3: An improved version of the algorithm in Listing 4.2. Each node is placed in the centre of their annulus wedge and no longer on the left. Otherwise, the algorithms are identical, and every sibling has the same amount of space.

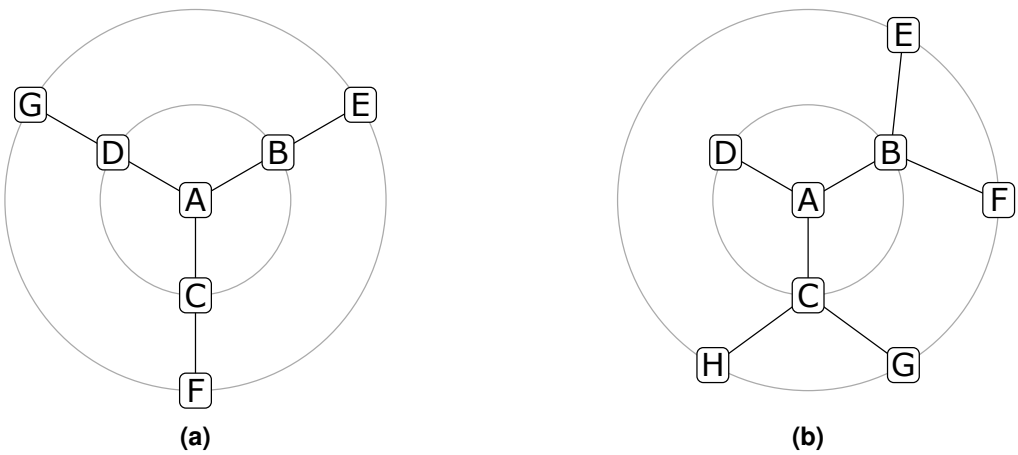


Figure 4.6: Displaying the same hierarchies as in Figure 4.5, only that this time nodes are centred in their annulus wedge, which leads to pleasing results for both hierarchies. [Diagrams generated by the author of this thesis using the Python programming language [Python, 2014].]

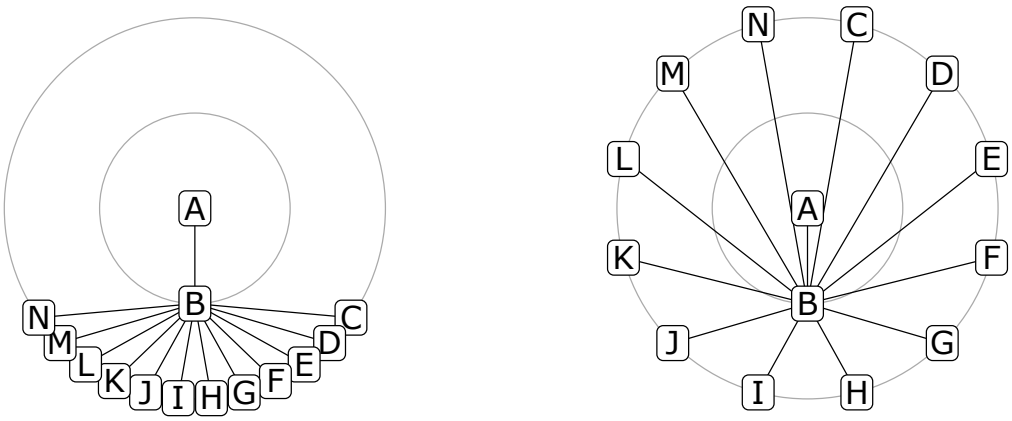


Figure 4.7: Visualising a hierarchy with the annulus wedge restricted (left) and with the wedge unrestricted (right). In the unrestricted version of (b), with the algorithm shown in Listing 4.3, edges escape the ring the nodes are on and cross the circle the parent node is on. Depending on the hierarchy and the precise algorithm, multiple circles could be crossed. To avoid such unaesthetic and irritating crossings, the annulus wedge can be restricted by a maximum arc that makes such crossings impossible, as happens in (a). To restrict the annulus wedge, the algorithm of Listing 4.3 is modified to use Equation 4.6, which leads to the algorithm in Listing 4.5. [Diagrams generated by the author of this thesis using the Python programming language [Python, 2014].]

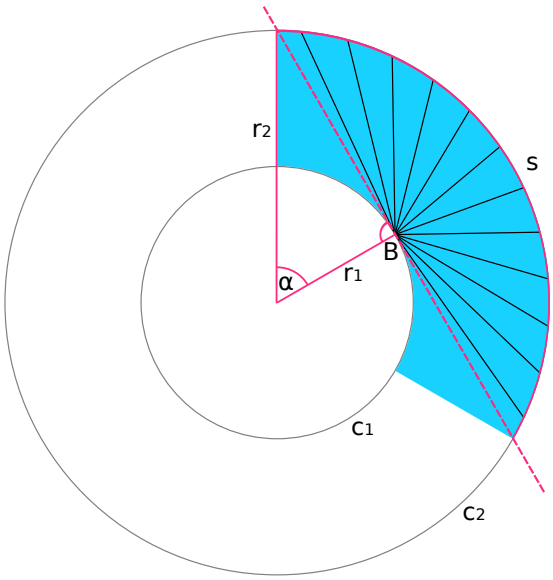


Figure 4.8: A method to limit the annulus wedge. A tangent line (dashed) is drawn at node B. The maximum limit of the annulus wedge is determined by the circle segment s, determined by the intersections of the tangent at B with circle c_2 where the child nodes are. Calculating the magnitude of s is easy, as the relationship $s = 2 * \alpha$ exists in polar coordinates. [Diagram drawn by the author of this thesis using the Python programming language [Python, 2014] and Inkscape [Inkscape, 2014].]

Level	$4 * asin$	$2 * acos$
1	120°	120°
2	167°	96°
3	194°	83°
4	213°	74°
5	226°	67°
6	236°	62°
7	244°	58°

Table 4.1: The effects of using the wrong equation to limit the annulus wedge are shown. The second column uses the wrong Equation 4.1, while the third column uses the correct Equation 4.6. The annulus wedge calculated for level 1 has the same results in both cases. After that, the results using the wrong equation increase instead of decreasing.

determined by the intersections of the tangent at node B with the circle c_2 the children are on. Book and Keshary [2001] suggest to determine s as

$$s = 4 * asin\left(\frac{r_1}{r_2}\right) \tag{4.1}$$

which is actually incorrect, as is proven below.

From trigonometry it follows that:

$$\alpha = acos\left(\frac{r_1}{r_2}\right) \tag{4.2}$$

moreover, an attribute of polar coordinates is

$$s = 2 * \alpha \tag{4.3}$$

Combining Equations 4.2 and 4.3 results in

$$s = 2 * acos\left(\frac{r_1}{r_2}\right) \tag{4.4}$$

which is different from Equation 4.1. Choosing the wrong equation to limit the annulus wedge leads to incorrect results, as can be seen in Table 4.1. Instead of decreasing, the annulus wedge increases.

Figure 4.9b shows the effects of the incorrect equation on fixed-wedge layout. All numbered nodes are placed in the annulus wedge of D, which inherited an annulus wedge of 120° from A: 120° ≤ 194°, which would be the maximum annulus wedge according to the incorrect Equation 4.1. Instead D should have an annulus wedge of 83°, shown in Figure 4.9a. Book and Keshary [2001] used a flexible-wedge layout, discussed in Section 4.2, and there the effects are devastating, as can be seen in Figure 4.10b. This devastating effect of Equation 4.1 cannot be observed in Book and Keshary [2001], since neither deep nor sparse hierarchies are used.

Equation 4.4 can be generalised, where l is the level for which to calculate the maximum annulus wedge:

$$s_l = 2 * acos\left(\frac{r_l}{r_{l+1}}\right) \tag{4.5}$$

Moreover, if r_i is based on a constant R multiplied with the level l, then Equation 4.5 can be changed to

$$s_l = 2 * acos\left(\frac{l}{l+1}\right) \tag{4.6}$$

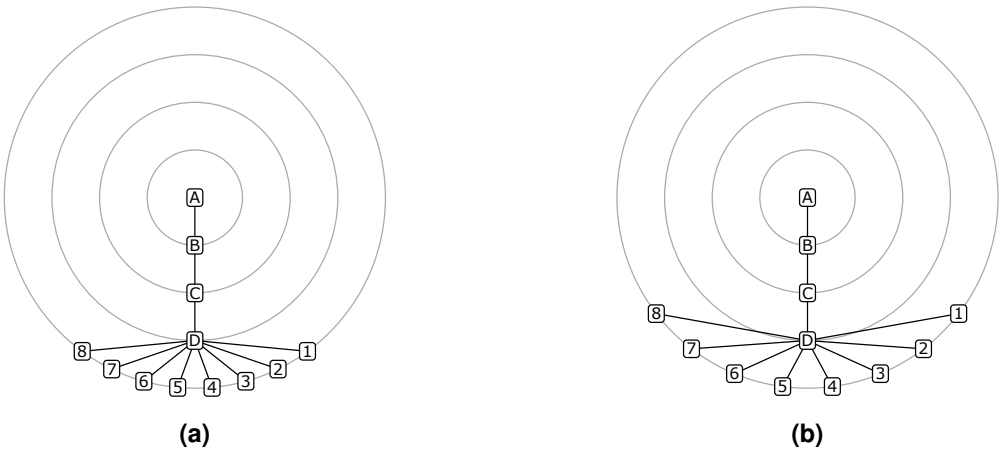


Figure 4.9: The annulus wedge is determined correctly in (a). In (b), the edges of nodes 1 and 8 cross the circle their parent is on, because of using the incorrect Equation 4.1. [Diagrams generated by the author of this thesis using the Python programming language [Python, 2014].]

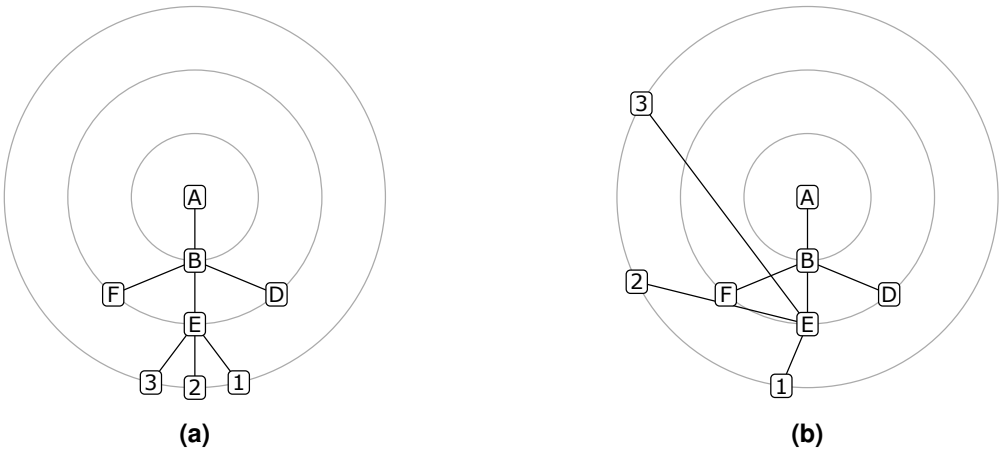


Figure 4.10: The annulus wedge is determined correctly in (a). The incorrect Equation 4.1 is used in (b), leading to displeasing results for flexible-wedge layout. [Diagrams generated by the author of this thesis using the Python programming language [Python, 2014].]

```
function applyRestrictions(node, leftLimit, arcLength)
  maxArcLength = 2 * acos(node.level / (node.level + 1))

  if arcLength <= maxArcLength
    return (leftLimit, arcLength)
  else
    adaptedArcLength = min(arcLength, maxArcLength)
    adaptedLeftLimit = leftLimit + (arcLength - maxArcLength) / 2
    return (adaptedLeftLimit, adaptedArcLength)
  end
end
```

Listing 4.4: The function restricts the given annulus wedge denoted by `leftLimit` and `arcLength`. The resulting annulus wedge is not larger than the maximum wedge. Moreover, restricted annulus wedges are centred in the maximum annulus wedge.

where s_l is the maximum annulus wedge for level l. The final annulus wedge is the minimum of s_l and the available space per node. If there are many siblings, each node receives less space than the maximum annulus wedge for the current level.

Restricting the annulus wedge leads to a new question: What happens with remaining arc length? There are multiple possibilities:

- Place the restricted annulus wedge inside the original annulus wedge and continue with the other neighbouring annulus wedges as if there were no restriction. Thus the neighbouring annulus wedge will be placed on the right of the original annulus wedge. This will lead to empty space in the original annulus wedge.
- Left align the restricted annulus wedge inside the original annulus wedge and place the neighbouring annulus wedge directly to the right of the restricted annulus wedge. There will be empty space between the first and last annulus wedges.
- Divide the remaining arc length between the other annulus wedges. For any arc length still remaining use one of the other strategies. This leads to space efficient layouts, but is more complicated to perform.

The first approach is used in Figure 4.7a. The algorithms described so far take the left of the annulus wedge `leftLimit` and the arc length `arcLength` of the annulus wedge. To centre the restricted wedge inside the original, both `arcLength` and `leftLimit` have to be adapted, which is shown in Listing 4.4. First, the maximum arc length is calculated using Equation 4.6. Then the aforementioned parameters are adapted if necessary. The following listings directly refer to the function `applyRestrictions` from Listing 4.4. Listing 4.5 uses `applyRestrictions` with the previous algorithm in Listing 4.3. In contrast to Book and Keshary [2001], the nodes are also centred inside the function `applyRestrictions`.

4.1.3 Determining the Size of the Annulus Wedge

So far, the issues of annulus wedge restriction and positioning nodes inside their own annulus wedge were discussed, while the question of how to determine the size of the annulus wedge was not investigated in more detail. The annulus wedge size was set to be the same for each sibling by simply dividing the parent's annulus wedge by the number of children. This may lead to wasted space though. For example, space might be wasted when every child node receives the same share of the annulus wedge, no matter

```
function drawChildNodes(parentNode, leftLimit, arcLength)
  for child in parentNode.children
    childArcLength = arcLength / parentNode.numChildren
    child.azimuth = leftLimit + childArcLength / 2

    adaptedLeftLimit, adaptedChildArcLength =
        applyRestrictions(child, leftLimit, childArcLength)

    draw(child, parentNode)
    drawChildNodes(child, adaptedLeftLimit, adaptedChildArcLength)

    leftLimit = leftLimit + childArcLength
  end
end
```

Listing 4.5: In this algorithm, based on Listing 4.3, the annulus wedge is restricted using Equation 4.6, so as to avoid edges crossing circles.

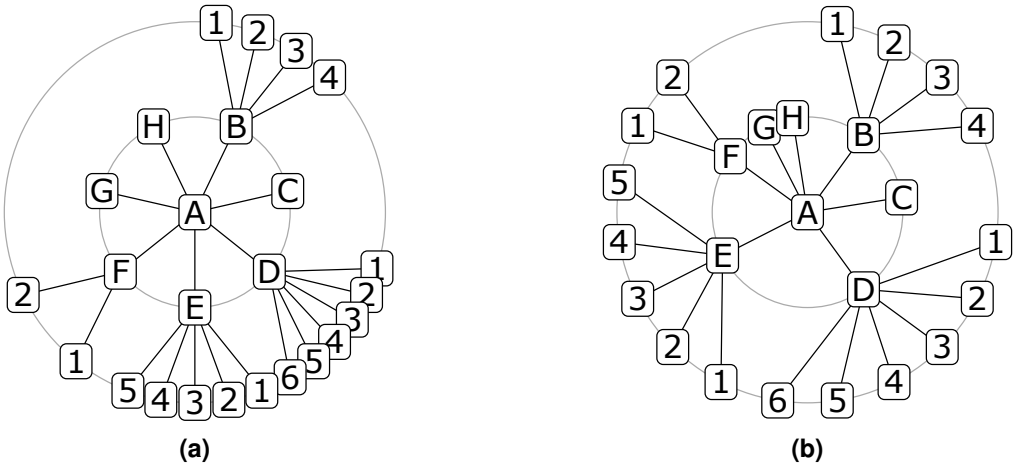


Figure 4.11: The same hierarchy is visualised twice. In (a), every sibling receives the same amount of space. This leads to the child nodes of D being very close together, while child nodes of other nodes like F have more space. In contrast, in (b), the annulus wedge for nodes on the first level is determined by the share they have of all child nodes on the second level using the algorithm in Listing 4.6. As a result, all nodes on the second level have the same amount of space and the child nodes of D have more space than before. [Diagrams generated by the author of this thesis using the Python programming language [Python, 2014].]

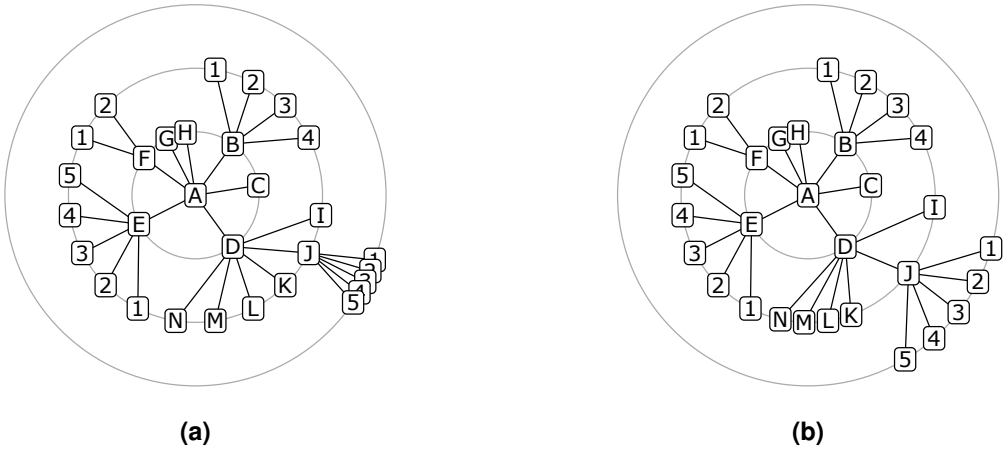


Figure 4.12: In (a), the annulus wedge for nodes on the first level is determined by the share they have of all child nodes on the second level, while on the right the wedge size on all levels depends on the child count. This leads to the different layouts of nodes below node D. In (a), each child of node D has the same space, leading to little space for the children of node J. In contrast, the children of node J have more space in (b). Notable are the changes for the siblings of node J, which not only have less space on the right, but they also seem a little disconnected of node J. This highlights the trade-offs that need to be considered. The choice of algorithm depends both on the target hierarchy and the use case. [Diagrams generated by the author of this thesis using the Python programming language [Python, 2014].]

how many children or descendants it has, as is the case in Figure 4.11a. Moreover, in that case, nodes tend to be very close at lower levels making it impossible to discern them.

The layout in Figure 4.11a is not very space efficient, since every sibling has the same amount of space. A more space-efficient way in this case would be to consider the number of descendants when deciding on the annulus wedge size. The following algorithms differ in the type of descendants that are considered. When determining the wedge size of a node, some algorithms only consider direct descendants (children), while others consider more descendants.

Sheth and Cai [2003] describe an algorithm where the annulus wedge's size depends both on the level and on the number of children of a node. The annulus wedge for the nodes on level 1 depends on the number of children they have. A node on level 1 that has more children than its siblings will receive a larger share of the available 360°. Therefore, nodes on level 1 do not have the same amount of space, while every node on level 2 receives the same space, as shown in Figure 4.11b. Wedge restrictions can change this property though. For all other levels nodes receive the same amount of space as their siblings, ignoring the number of children they have.

To calculate the wedge size for the nodes on level 1, first the number of children it and all its siblings have needs to be calculated. This could be rephrased to counting the number of grandchildren their direct parent has, which is the root node, like `countGrandchildren(rootNode)` in Listing 4.6. Nodes having no children are counted as if they had one, since otherwise edges would overlap. The available space is then divided by this count and assigned to nodes on level 1 depending on their number of children. Listing 4.6 contains both the algorithm to count grandchildren and the algorithm to draw Figure 4.11b.

From Listing 4.6 it is very easy to implement an algorithm where the wedge size is always based on the child count, as is discussed in Battista et al. [1999, pages 52–53]. The only difference in Listing 4.7 is that the algorithm is less complicated now, as `drawChildNodes` calls itself recursively instead of another function. Figure 4.12 compares the original algorithm with the simplified one.

All descendants are considered for annulus wedge determination in Listing 4.8. The hierarchy is harder to discern, as shown in Figure 4.13, since nodes with many descendants receive more space than other nodes, ignoring the level the nodes are on. That means every descendant has the same weight, even

```
function drawChildNodes(parentNode, leftLimit, arcLength)
  arcPerGrandChild = arcLength / countGrandchildren(parentNode)
  for child in parentNode.children
    childArcLength = max(child.numChildren, 1) * arcPerGrandChild
    child.azimuth = leftLimit + childArcLength / 2

    adaptedLeftLimit, adaptedChildArcLength =
        applyRestrictions(child, leftLimit, childArcLength)

    draw(child, parentNode)
    drawChildNodesAfterFirstLevel(
        child,
        adaptedLeftLimit,
        adaptedChildArcLength)

    leftLimit = leftLimit + childArcLength
  end
end

function countGrandchildren(node)
  result = 0
  for child in node.parentNode
    result = result + max(child.numChildren, 1)
  end
  return result
end

function drawChildNodesAfterFirstLevel(parentNode, leftLimit, arcLength)
  ...
end
```

Listing 4.6: In this algorithm, which is based on Sheth and Cai [2003], the annulus wedge size for nodes on the first level depends on the number of child nodes they have. The more child nodes, the larger the annulus wedge. Nodes after the first level are treated equally, like in the previous algorithm. Thus `drawChildNodesAfterFirstLevel` refers to `drawChildNodes` of Listing 4.5.

```
function drawChildNodes(parentNode, leftLimit, arcLength)
  widthPerGrandChild = arcLength / countGrandchildren(parentNode)
  for child in parentNode.children
    childArcLength = max(child.numChildren, 1) * widthPerGrandChild
    child.azimuth = leftLimit + childArcLength / 2

    adaptedLeftLimit, adaptedChildArcLength =
        applyRestrictions(child, leftLimit, childArcLength)

    draw(child, parentNode)
    drawChildNodes(child, adaptedLeftLimit, adaptedChildArcLength)

    leftLimit = leftLimit + childArcLength
  end
end
```

Listing 4.7: Simplified version of the algorithm in Listing 4.6. Now `drawChildNodes` calls itself recursively instead of another function. The wedge size of a `child` of the current `parentNode` depends on the number of children it has.

if located many levels below the current node. A workaround could be to assign different weights to descendants, depending on the distance to the current node, which is assigned to future work.

```
function drawChildNodes(parentNode, leftLimit, arcLength)
  if parentNode.numChildren == 0
    return
  end

  sharePerDescendant = arcLength / (parentNode.numDescendants - 1)
  for child in parentNode.children
    childArcLength = sharePerDescendant * child.numDescendants
    child.azimuth = leftLimit + childArcLength / 2

    adaptedLeftLimit, adaptedChildArcLength =
        applyRestrictions(child, leftLimit, childArcLength)

    draw(child, parentNode)
    drawChildNodes(child, adaptedLeftLimit, adaptedChildArcLength)

    leftLimit = leftLimit + childArcLength
  end
end

function countDescendants(node)
  count = 1
  for child in node.children
    count = count + countDescendants(child)
  end
  node.numDescendants = count
  return count
end
```

Listing 4.8: The annulus wedge is based on the number of descendants, instead of only children. `countDescendants` has to be called on the root node before calling `drawChildNodes`. For each node, it counts the number of descendants, including the node itself, and makes that value accessible via `node.numDescendants`.

(a) **(b)**

Figure 4.13: In (a), the annulus wedge is determined by the number of children a node has. In contrast, the number of descendants is relevant in (b). As a result, some nodes that have no children, like nodes G and L, are barely visible in (b), since other nodes with many descendants receive more space. The hierarchy is harder to discern, especially when the number of descendants varies strongly between siblings. [Diagrams generated by the author of this thesis using the Python programming language [Python, 2014].]

4.2 Flexible-Wedge Layouts

The layout in Figure 4.14 illustrates a common issue: there is much unused space, since many nodes have no children. Of the outer circle, less than half the space is used. One solution was already proposed by assigning space to nodes based on the number of children they have, as shown in Figure 4.15. A disadvantage of this approach is that siblings now have different amounts of space available, which changes the appearance of the hierarchy: nodes B, C and D are very close together which suggests they may comprise a group. However, there is no group, the only commonality in this case is that these nodes have no children and are neighbours. The results can be even worse for huge hierarchies, where leaf nodes can be grouped so close to each other that they are indistinguishable. This affects comprehension of the hierarchy, as it creates an artificial focus on nodes with many children and discriminates against those with few or no children.

Book and Keshary [2001] tried to solve this problem by relaxing the annulus wedge restriction. Child nodes are no longer strictly bound to the annulus wedge of their parent, but can also use any available space which is unused by their cousin nodes. To avoid overlapping edges, nodes still have to comply to the maximum annulus wedge discussed in Section 4.1.2. In Figure 4.16, a simplified and extended version of the algorithm described by Book and Keshary [2001] is used: The nodes are centred in their assigned area and the calculation of the maximum annulus wedge is corrected, as described in Section 4.1.2. Compared to Figure 4.15 there is roughly the same amount of empty space left, although that highly depends on the hierarchy. An advantage of this algorithm compared with fixed-wedge layouts is that empty space at higher levels can be used, whereas the annulus wedge of fixed-wedge layouts becomes smaller the higher the level. Nonetheless, a disadvantage of this layout is that it can appear rather overcrowded. Moreover, empty space can actually help in the comprehension of hierarchies. The space distribution can also lead to confusion since it is only based on the directly available space. Some nodes might receive less space while others receive more than they can use. This can make comprehension of the hierarchy harder and lead to misconceptions.

The algorithm is more complicated than the previous ones, since an iterative approach is taken and more factors have to be considered. Moreover, having the left limit and the arc length is no longer enough information. Instead, both `leftLimit` and `rightLimit` are directly stored in the nodes and are 0 by default. Listing 4.9 contains helper functions for layout calculation:

- `normaliseArc` normalises the arc to the range $[0,\ 2 * \pi)$.
- `calculateArcLength` calculates the length of the arc that ranges from `left` to `right`. If the arc spans the whole circle, $2 * \pi$ is returned.
- `angleDist` calculates the distance between the given angles. The distance is not normalised and thus can be negative. This is useful to determine whether `left` is really left of `right`.
- `calculateLimitToNext` calculates the annulus wedge limits between the given nodes. If the maximum annulus wedges of both do not cross, then these are assigned as limits. Otherwise, the point where the wedges cross is determined as the limit.

These functions are then used to calculate the layout which is done by the functions in Listing 4.10:

- `drawChildNodes` is the entry function of the algorithm as before. The algorithm is split into three steps:
  1. Draw all child nodes of the parent nodes collected before, which draws all nodes of the current level. The initial parent node is the root node if it has children.
  2. Collect all nodes on the current level which have children, they are the parent nodes for the next run.

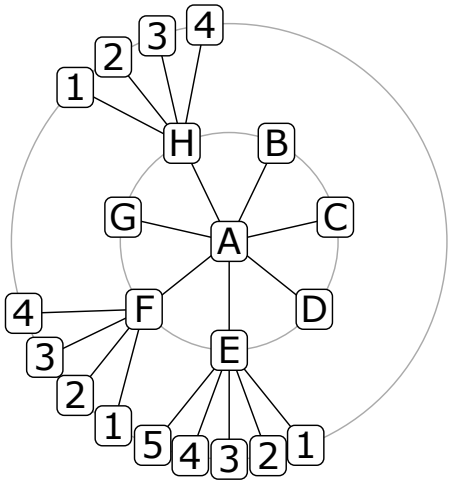


Figure 4.14: There is unused space in the visualisation of the hierarchy. Nearly half of the outer circle is unused. The reason for that is the algorithm, which gives every sibling the same amount of space and forces child nodes to be placed in the annulus wedge of their parent node. [Diagram generated by the author of this thesis using the Python programming language [Python, 2014].]

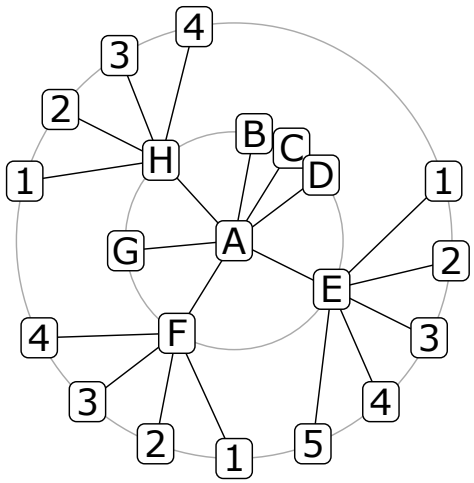


Figure 4.15: Compared to Figure 4.14, less space remains unused. The annulus wedge is determined by the number of children a node has. Therefore, siblings need not have the same amount of space anymore. A disadvantage of this approach is that connections might be visually suggested which do not exist: Nodes B, C and D are very close together and appear to be grouped even though their only commonality is that they have no child nodes and are neighbours. For huge hierarchies, such leaf nodes might be placed so close together that they are indistinguishable. This can make it harder to understand hierarchies by creating an artificial focus on densely packed nodes. [Diagram generated by the author of this thesis using the Python programming language [Python, 2014].]

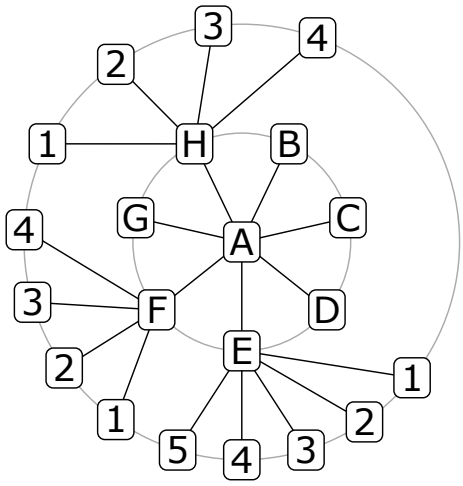


Figure 4.16: A flexible-wedge layout. Every sibling has the same amount of space. They can use free space next to them. To ensure that no edge crossings appear the maximum annulus wedge is used in this algorithm as well. Naturally, this algorithm also has weaknesses. For one, it can lead to layouts which appear overcrowded. Moreover, the space distribution solely depends on the location of the nodes. There is less empty space around node F. Only its neighbour G has no children and that space has to be shared with nodes F and H. In contrast, both nodes E and H can utilise much empty space for their children, since nodes B, C and D have no children. Indeed, nodes B, C and D have even more space available than they can use. Depending on the hierarchy, some nodes might receive very little space, while others receive much more space, resulting in a layout which could lead to misconceptions. [Diagram generated by the author of this thesis using the Python programming language [Python, 2014].]

3. For these nodes calculate the annulus wedges. Increase the looked at level and go to step one. Stop if the maximum level is reached.

The function `findMaxLevel` finds the maximum level of the hierarchy and is left out for brevity. A depth-first search can be used to do that.

- `layoutChildren` lays out the children of the given node. It is very similar to previous algorithms, but does not use recursive calls. Moreover, child nodes which are parents themselves are added to the `parentsOut` list.

- `calculateWedges` calculates the wedges for the given nodes. Notice that `drawChildNodes` calls this function only with nodes which themselves have children. Therefore, nodes having no children are ignored and their space can be used.

4.3 Adapting Layered Node-Link Tree Visualisations to Radial Node-Link Tree Visualisations

Two-dimensional layered node-link tree visualisations can be adapted to create radial tree visualisations by projecting the layout radially. This can be done by adapting the Cartesian coordinates of each node to polar coordinates, following some rules described below.

Instead of the y-coordinate of a node, the distance from the centre is used, which is set to the level of the node. The azimuth γ of a node can be calculated with Equation 4.7. The x-coordinate x of the node, the smallest x-coordinate x_{min} and the largest x-coordinate x_{max} of the nodes in the original layout need to be known.

$$\gamma = \frac{2\pi}{x_{max} - x_{min}} * (x - x_{min}) \tag{4.7}$$

```
function normaliseArc(angle)
  return angle % 2 * PI
end

function calculateArcLength(left, right)
  length = right - left
  if length == 2 * PI
    return length
  else
    return normaliseArc(length)
  end
end

function angleDist(left, right)
  left = normaliseArc(left)
  right = normaliseArc(right)

  dist = right - left
  if dist > PI
    return dist - 2 * PI
  elif dist < -PI
    return dist + 2 * PI
  else
    return dist
  end
end

function calculateLimitToNext(node, nextNode, maxArc)
  node.rightLimit = normaliseArc(node.azimuth + maxArc / 2)
  nextNode.leftLimit = normaliseArc(nextNode.azimuth - maxArc / 2)

  if node == nextNode:
    return
  end

  dist = angleDist(node.rightLimit, nextNode.leftLimit)
  if dist >= 0 or -dist >= maxArc:
    return
  end

  node.rightLimit = normaliseArc(node.rightLimit + dist / 2)
  nextNode.leftLimit =  normaliseArc(nextNode.leftLimit - dist / 2)
end
```

Listing 4.9: Helper functions for flexible-wedge layout calculation.

```
function drawChildNodes(rootNode, leftLimit, arcLength)
  rootNode.leftLimit = leftLimit
  rootNode.rightLimit = arcLength

  parents = []
  if rootNode.numChildren > 0
    parents.append(rootNode)
  end

  maxLevel = findMaxLevel(rootNode)
  for level in range(1, maxLevel + 1)
    tempParents = []

    for parent in parents
      layoutChildren(parent, tempParents)
    end

    parents = tempParents
    calculateWedges(parents, level)
  end
end

function layoutChildren(parentNode, parentsOut)
  arcLength = calculateArcLength(parentNode.leftLimit, parentNode.
      rightLimit)
  childArcLength = arcLength / parentNode.numChildren
  leftLimit =  parentNode.leftLimit

  for child in parentNode.children
    child.azimuth = leftLimit + childArcLength / 2
    leftLimit += childArcLength

    if child.numChildren > 0
      parentsOut.append(child)
    end
  end
end

function calculateWedges(parents, level)
  maxArc = 2 * acos(level / (level + 1)))
  for i, node in enumerate(parents)
    nextIndex = (i + 1) % parents.numChildren
    nextNode = parents[nextIndex]
    calculateLimitToNext(node, nextNode, maxArc)
  end
end
```

Listing 4.10: Functions to calculate a flexible-wedge layout. The helper functions in the previous listing are used.

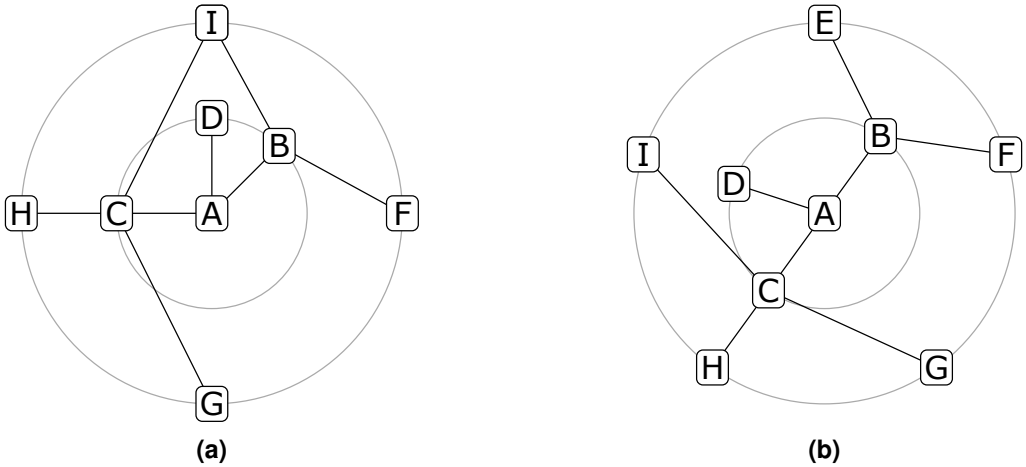


Figure 4.17: Both figures show hierarchies visualised with the Walker algorithm adapted to radial tree layouts. In (a), node E is hidden by node I, because the original layout needs no separation between the left and right contour of the hierarchy. To fix this issue, in (b), it is ensured that the left and right contour of the hierarchy are separated. Edges escape the ring the nodes are on in both visualisations, which is a disadvantage when adapting the Walker algorithm. [Diagrams generated by the author of this thesis using the Python programming language [Python, 2014].]

With these small changes, the Walker [Walker, 1990] algorithm can be adapted to radial trees as is discussed in Nussbaumer [2005]. The Walker algorithm does not use the annulus wedge concept. Still, the outline of each subtree can be thought of a flexible annulus wedge, which is not allowed to overlap with any other annulus wedge. Figure 4.17a shows the Walker algorithm adapted to the hierarchy of Figure 3.1 on page 7. There are some obvious issues:

- There are no node placement restrictions, as long as the rules described above are met. As a result, edges can escape the ring the nodes are on, which is a disadvantage when adapting the Walker algorithm. The influence of this effect depends on the structure of the hierarchy: very visible in sparse hierarchies, but hardly recognisable in dense ones.
- Node B has a child node E which is placed behind node I. The Walker algorithm takes care that every node has at least the aforementioned distance d to other nodes on their right. In the original layout node E is not on the right of node I, but when adapting the layout it is. To fix this issue, x_{max} of Equation 4.7 needs to be increased by a value ensuring that the left contour of the original layout has at least distance d to the right contour of the original layout, as is shown in Figure 4.17b.

There can be be disadvantages when adapting algorithms, such as those discussed for the Walker algorithm above. Moreover, the change needed for x_{max} depends on the adapted algorithm and cannot be considered independently, because otherwise the visualisation might look inconsistent. More in-depth work is needed for algorithms where just changing x_{max} is not enough to create a consistent visualisation.

Chapter 5

Implementing Node-Link Radial Trees for HVS

The radial tree browser is the implementation of radial tree support in HVS [Andrews, Putz, and Nussbaumer, 2007]. The main goal was to provide a testbed for many different radial tree layouts, allowing users to compare algorithms with hierarchies of their choice. Another goal was to support interactivity.

5.1 Hierarchical Visualization System (HVS)

HVS is a flexible framework created by Putz [2005] and Nussbaumer [2005] for visualising hierarchies. Any visualisation possible with Java [Oracle, 2014], which is the language HVS was written in, can be realised. This versatility is highlighted by the visualisations already present:

- Cone Explorer, which uses cone trees. Unfortunately Cone Explorer is no longer working with newer Java versions.
- Dendrogram browser.
- Hyperbolic browser, which uses hyperbolic geometry and a radial tree [Nussbaumer, 2005, pages 51–62].
- InfoLens browser, which projects a radial tree layout onto an equatorial disc [Nussbaumer, 2005, pages 73–83]. A fish-eye distortion is provided to have a magnified focus area.
- Information Pyramids [Putz, 2005, pages 59–72], which utilises a three-dimensional technique with plateaus as nodes. These plateaus are stacked upon each other to visualise the hierarchy.
- Magic Eye browser, which is similar to the Hyperbolic browser but uses spherical projection [Nussbaumer, 2005, pages 63–72].
- Sunburst browser.
- Tree View and TreeView+ browsers which use an outliner style orthogonal tree representation.
- Treemap Explorer.
- Walker Tree browser which uses classic trees to present the hierarchy [Nussbaumer, 2005, pages 41–50].

HVS supports interactivity, both within a visualisation and also between visualisations, by providing events via its application programming interface (API). Filter and search capability are also supported. Moreover, the underlying hierarchy is abstracted away in so-called data sources, to provide extensibility. The local file system can be used as a source hierarchy, but hierarchies stored in TreeML [TreeML, 2006; Fekete and Plaisant, 2003] format are also supported. Chapter 7 on page 49 describes the extension of HVS to support the import of hierarchies stored in SKOS [Miles and Bechhofer, 2009] file format.

5.2 Graphview Framework

Since radial trees are also graphs, the graph related foundations created by Nussbaumer [2005, pages 41–89] for HVS can be reused. The graphview framework hides away many details of HVS. Moreover, a specific tree structure `TreeGraph` is provided, which supports the storage of additional information in the `TreeNode` nodes. `TreeNode` itself is also graphview specific. It contains a HVS `Node` instance and further additional information related to the tree layout algorithms.

The graphview framework provides multiple classes which can be subclassed to create a new graph-based visualisation:

- `HvsView` acts like an abstract factory [Gamma et al., 1994, pages 87–95] where subclasses specify which building blocks are used to create the visualisation. Moreover, `HvsView` interacts with HVS. It can both fire and receive HVS events and automatically reacts to many of those, while subclasses can also react to events that are not handled.
- `GeometryEngine` is the base class for tree layouts. Subclasses are required to calculate the layout of the nodes, act on zoom and focus changes, and react to panning.
- `RenderEngine` draws the graph created by `GeometryEngine` on the screen. Subclasses only need to be created if specific behaviour is needed that is not present.
- `Settings` provides the menu items added by the visualisation.
- `InputHandler` receives mouse-related events like click or scroll events. These events can, for example, lead to a node being selected or the context menu being displayed.

The aforementioned `TreeNode` had many variable members which were highly algorithm-specific. To both reduce the overhead and make the code clearer, the author made it possible to create subclasses of `TreeNode` which work correctly with `TreeGraph` and the other related classes. So far `RTNode`, `HBNode` and `ILNode` were introduced which are the nodes used for radial tree layouts, the hyperbolic browser, and InfoLens respectively.

Before this thesis, `RenderEngine` drew to an image buffer and assumed that every `GeometryEngine` subclass did its drawing onto a 500 by 500 pixel canvas. Therefore all drawing commands were scaled from this fictional canvas to the actual image buffer size depending on the size of the window the visualisation is in. A grave disadvantage of such behaviour becomes apparent when the window is resized: The scale of both the nodes and the font changes. Depending on the resize operation, everything might appear larger or smaller or worse might also be stretched. Figure 5.1 shows a hierarchy visualised with InfoLens [Nussbaumer, 2005, pages 73–83]. Since the window was resized horizontally and InfoLens uses `RenderEngine` directly, the result is stretched and looks displeasing. Another disadvantage of scaling the canvas is that regardless of the window size, the same view will always be displayed, just at a different scale, thus resizing will not lead to a more detailed view.

The Walker layout browser circumvents these `RenderEngine` disadvantages by reimplementing most of `RenderEngine` in a subclass, even using duplicated variables for the same thing. However, creating a separate `RenderEngine` subclass for the radial tree browser was not an option, as it would have resulted

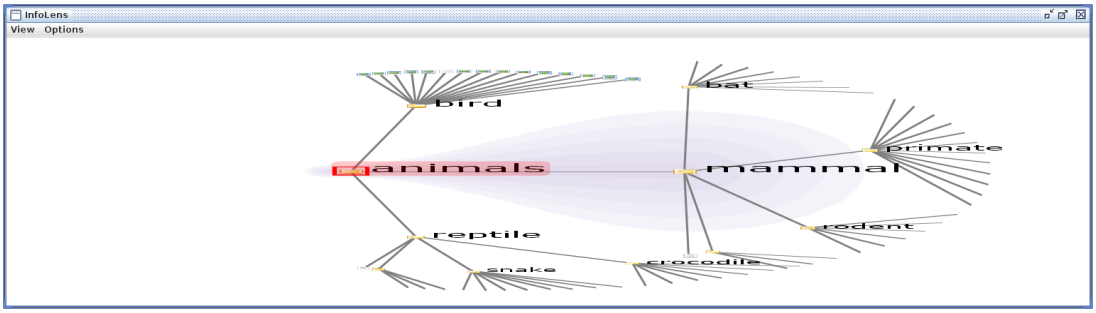


Figure 5.1: Resizing a window containing the InfoLens visualisation only scales the view. This leads to displeasing stretching of the view when the window is resized only in one dimension, like in this case only the x-axis. `RenderEngine` causes such behaviour by the way it handles drawing on the canvas. [Screenshot taken by the author of this thesis.]

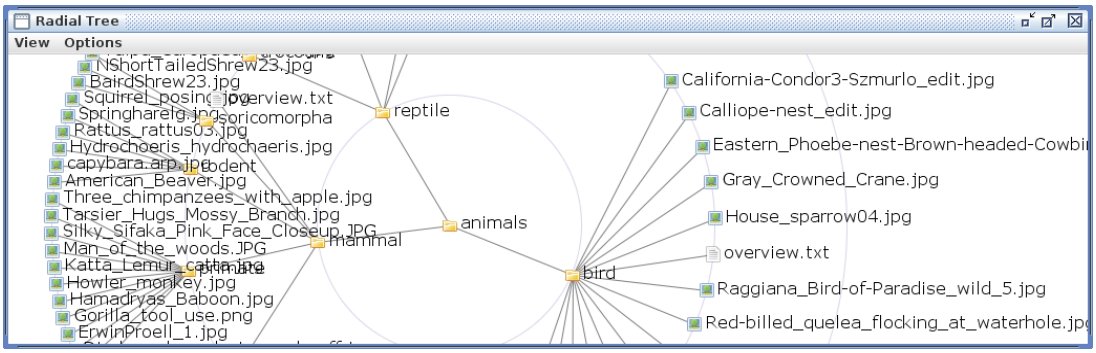


Figure 5.2: Resizing a window containing the radial tree visualisation changes the available space. In this case the view was zoomed initially and then resized on the x-axes giving it more space, resulting in more information being displayed at once. There is no stretching, thus the icons and the labels keep their previous size [Screenshot taken by the author of this thesis.]

in much code duplication. Therefore `RenderEngine` was extended to provide two modes: one where the canvas is scaled and one where visualisations have to handle layout adaptions themselves and thus can have a layout that is not scaled. Special care was taken to ensure that the behaviour of existing hierarchy browsers remained unchanged. The radial tree browser implements the new mode and as a result takes care of all layout adapations itself. For example, in Figure 5.2 there are no distortions caused when resizing the radial tree browser. Resizing affects neither node nor label size.

5.3 Layout Calculation

The `GeometryEngine` subclass `RTGeometryEngine` handles all the layouting and interaction with the graph as well as the animations. For clarity, many of these tasks are delegated to specific classes, such as the layouting itself, for which `RTLayout` with its subclasses is used. Most of the radial tree layouts discussed in Chapter 4 were also implemented for HVS.

Figure 5.3 illustrates the class hierarchy of the layout classes. The base classes `RTLayout` and `RTFixedWedgeLayout` do not represent actual layouts themselves, instead they contain common functionality. All the other classes represent specific layouts:

- `RTFlexibleWedgeLayout` uses the algorithm of Listing 4.10 on page 33.
- `RTWalkerLayout` adapts the Walker [Walker, 1990] algorithm for radial trees. Internally, it uses the `WalkerLayout` class introduced by Nussbaumer [2005] which does most of the work. The algorithm is adapted for radial trees using the approach discussed in Section 4.3 on page 31.

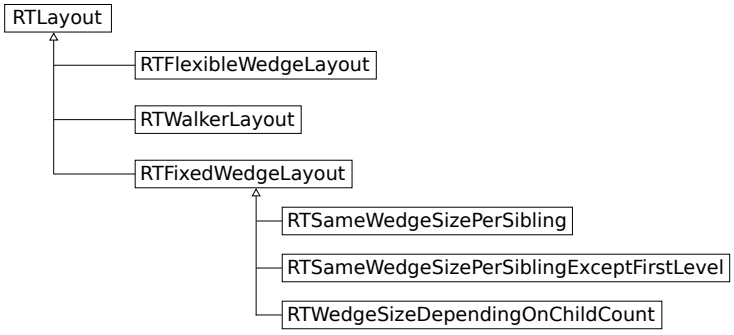


Figure 5.3: Hierarchy of the radial tree layout classes. The structure is similar to the classification in Section 4.1 and 4.2. [Diagram drawn by the author of this thesis using Inkscape [Inkscape, 2014].]

- `RTSameWedgeSizePerSibling` is based on Listing 4.5 on page 24.
- `RTSameWedgeSizePerSiblingExceptFirstLevel` utilises the algorithm in Listing 4.6 on page 26.
- `RTWedgeSizeDependingOnChildCount` uses the algorithm of Listing 4.7 on page 27.

The layouts each support different settings. Layout type is also a setting. Thus, layouts can be changed at any time. Appendix A on page 61 describes the available settings.

The layout process is split into two tasks which are also handled separately: calculating the layout and then adapting it. Subclasses of `RTLayout` determine the layout using a fictional canvas with fixed dimensions of 500 by 500. `RTLayout` on the other hand adapts the layout to the current canvas and the zooming level, and also does the coordinate transformation from the polar coordinate system to the Cartesian coordinate system. This separation of concerns leads to simplified code. As a result, a complete recalculation of the layout is needed in fewer cases, since changing the zoom level, or panning or resizing the window only needs an adaption of the existing layout, which improves performance.

As mentioned before, the layout calculated by subclasses of `RTLayout` assumes a fictional canvas. Therefore, this layout needs to be adapted to the current canvas by `RTLayout`, but without any stretching. In simplified form, the following steps are performed:

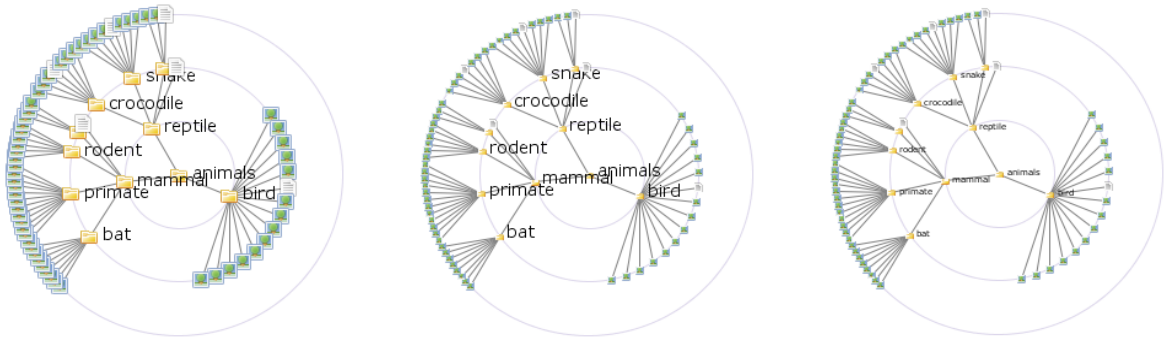
1. Apply the zoom level, which either reduces or increases the distance between nodes.
2. Transform the coordinate system from polar coordinates to the Cartesian coordinate system.
3. Move the centre of the view to reflect any panning done by the user.
4. Adapt the coordinates so that the centre of the view lies in the centre of the current window.
5. Crop the resulting image to the current window size.

With these steps, no unwanted stretching is possible.

5.4 Node and Label Size

`RenderEngine` can draw nodes either as an icon or as a coloured dot. Next to a node, there is an optional label, unless the nodes are too close to each other. Both the the node size and the font size are fixed and thus independent of the window size, the zoom level, or the visualised hierarchy.

Nevertheless, the radial tree browser supports optional downscaling of icon size and font size. When activated, the largest possible label size is the user configured font size. The icon size is determined based on the font size, but does not directly reflect it. Both nodes and labels can be downscaled from this



(a) Neither nodes nor labels are scaled.

(b) Only nodes are scaled.

(c) Nodes and labels are scaled.

Figure 5.4: Scaling labels and nodes. [Screenshots taken by the author of this thesis.]

size. If the size becomes too small, labels and nodes are no longer displayed. The hierarchy is then only visualised via edges.

To provide more flexibility, scaling of the font size and icon size were made independent of each other and can be turned on or off separately. Figure 5.4 shows an example. There is no scaling in the leftmost Figure (a), only nodes were scaled in the middle Figure (b), and both nodes and labels were scaled in the rightmost Figure (c).

The scaling is dependent on the density of the concentric circles, thus is only indirectly dependent on the zoom level. The closer the circles are to each other, either due to zooming or due to a deep hierarchy, the smaller both the nodes and labels become. The downscale factor ds is calculated using Equation 5.1, where r refers to the current radius of the inner circle on the screen. The lower limit of 0.3 ensures that the hierarchy will always be visualised, otherwise it would not be drawn on the screen anymore. The divisor of 90 was chosen by experiment.

$$ds = max\left(\frac{r}{90},\ 0.3\right) \tag{5.1}$$

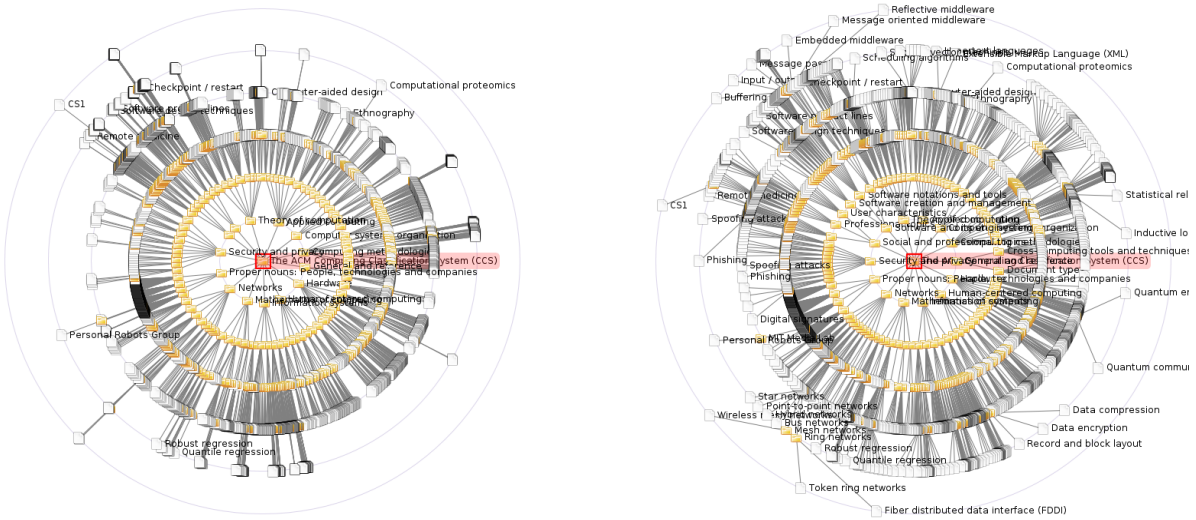
5.5 Navigation and Interaction

Navigating hierarchies and interacting with them is one of the key factors of information visualisation, which sets it apart from printed hierarchy diagrams. Especially for huge hierarchies, it is necessary to provide means for navigation and interaction with the hierarchy, to both better understand it as a whole, and to explore key parts of the hierarchy. The radial tree browser supports many of the interaction techniques discussed by Herman, Melançon, and Marshall [2000], as well as some others. These are described in the following subsections.

5.5.1 Zoom

Whenever the radial tree browser is started, it will display the whole hierarchy centred in the given space. The more levels a hierarchy has, the closer the circles are arranged in the view. With huge hierarchies like the one in Figure 5.5, the structure is hard to discern, regardless of which algorithm is used. Scaling the nodes and labels can help to make the hierarchy more discernible, although this would only help a little in the given example. Both nodes and edges are too close to each other.

The radial tree browser supports zooming to ameliorate this problem. In Figure 5.6, the user has zoomed in on the root node of the hierarchy. Zooming is done by placing the mouse cursor at the centre



(a) (b)

Figure 5.5: Visualising the ACM Computing Classification System [ACM, 2012] with two different radial tree algorithms. In (a), the annulus wedge is fixed and based on the child count. The algorithm of Listing 4.7 on page 27 is used. In contrast, in (b), the wedge is flexible, using the algorithm discussed in Section 4.2 on page 29. The flexible-wedge layout uses more of the available screen space, although it does not help much in discerning the hierarchy in this case. The edges and nodes are too close to each other in both layouts. [Screenshots taken by the author of this thesis.]

of the desired zoom location and then using the scroll wheel of the mouse. When zooming in, more details are displayed, like the labels of nodes that were too closely arranged before and are now further apart due to the zooming. When zooming out, fewer labels are displayed as the nodes become closer. Both node size and label size remain unaffected by zooming, unless otherwise configured as described in Section 5.4.

5.5.2 Pan

The radial tree browser also supports panning, which is essential in combination with zooming and thus often mentioned together [Herman, Melançon, and Marshall, 2000, page 33]. The view can be panned by dragging while keeping the left mouse button pressed. The location under the mouse cursor remains the same during panning, which makes panning intuitive-to-use. Pan limits are in place to ensure at least partial visibility of the hierarchy.

5.5.3 Fan-Out

Keith Andrews, the advisor of this thesis, suggested adding fan-out support for the focused (selected) node. Fan-out in this regard means to increase the wedge size of the focused node. If the focused node has no children, then its siblings also receive more space. As a result, the focused node and its children are easier to investigate while all the other nodes have less space. Figure 5.7 shows an example before and after a subtree was fanned out. Fan-out can be considered a focus+context [Herman, Melançon, and Marshall, 2000, page 34] technique, since context is maintained while specific parts are focused. There are multiple ways to trigger fan-out:

- Using the context menu on a node and selecting the fan-out action.
- Turning the mouse wheel while the control key is pressed changes the fan-out level of the focused node.

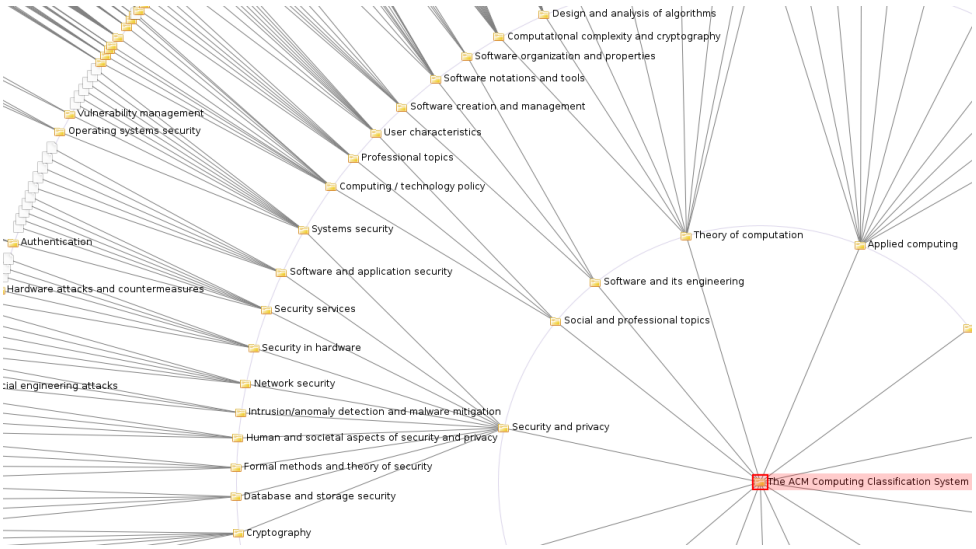


Figure 5.6: The same hierarchy as in Figure 5.5 after the user zoomed in to see more details of some nodes. [Screenshot taken by the author of this thesis.]

- Automatic fan-out is optionally activated whenever a focus event is retrieved, as discussed in the following section.

5.5.4 Animated Change of Focus

Whenever a radial tree browser receives a focus event, specific focus+context [Herman, Melançon, and Marshall, 2000; van Wijk and Nuij, 2004] animations are used to move and highlight the focus node. Two possible animations are currently implemented: fan-out and pan-to. These animations can be activated or deactivated in the settings. There are two ways to trigger a focus event:

- Single-clicking a node initiates a focus event for that node in every view, except the one where the node was clicked.
- Double-clicking a node initiates a focus event for that node in every view.

The first animation is fan-out. Any previously fanned out subtree is first smoothly fanned in again, then the subtree of the new focus node is fanned out. Cubic easing [Penner, 2002, pages 211–212] is used as an easing function. The amount of time used for an animation depends on the fanning level: when a subtree is hardly fanned out, the fan-in animation will take less time and vice versa. To reduce visual distraction, fan-out and fan-in are not always animated. For instance, if a sibling node is focused, no animation is needed as the result would not change much.

The second animation, pan-to, pans to the focused node. To have more context and to have fast animations, especially when the user zoomed in a long way, zooming is also involved during panning. At the beginning of panning, there is a zoom out and then later there is a zoom in again, according to the suggestions by van Wijk and Nuij [2003]; van Wijk and Nuij [2004]. They describe a model which supports very smooth zooming and panning. When the target node is already in view, there is no animation though, to keep distraction by movement to a minimum.

The animations were first realised with the Universal Tween Engine [Ribon, 2012] which provides a simple and flexible API. Unfortunately, there were problems when multiple views were animated at the same time: animations did not play at all or only with stuttering. Using a shared resource secured by locks did not improve the situation. Therefore, a simple animation framework was created which did not exhibit these issues.

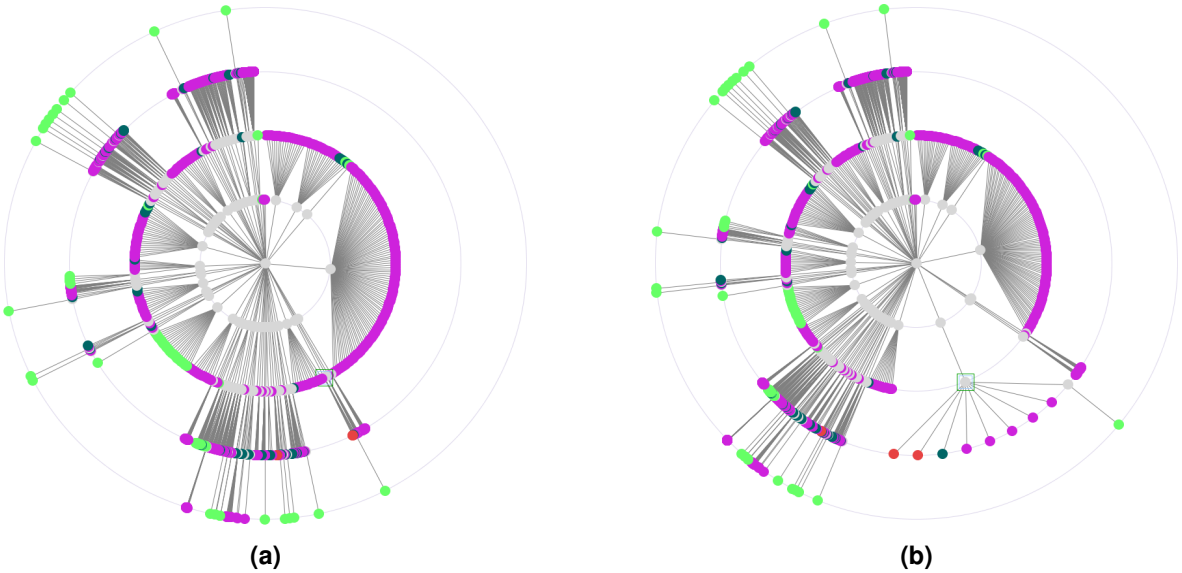


Figure 5.7: Fanning out a subtree of the hierarchy in (a) leads to the modified layout in (b). [Screenshots taken by the author of this thesis.]

5.5.5 Hierarchy Modification

The hierarchy can also be modified: nodes can be renamed, inserted, and removed. All of this functionality is provided by HVS and described in Putz [2005]. Due to the structure of HVS, visualisations simply inherit this functionality.

5.5.6 Opening Related Handler Programs

Upon middle-clicking a node, any associated handler program will be opened for that node. For example, if the node represents an image, an image viewer will be opened. This was limited to Windows for older versions of HVS. Now, most other platforms work as well including Linux and Mac OS X.

5.6 Outlook and Further Work

The fan-out animation exhibits some stuttering when an annulus wedge is reduced in size, due to the wedge size restrictions. This could be improved by animating each node separately as described by Yee et al. [2001]. Other experiments could be to make the radius of the circles more flexible and to have the node size dependent on criteria like the number of children. Moreover, a mode could be provided which chooses radial tree layouts automatically based on the current hierarchy. Currently, not every layout supports all settings, for example `RTFlexibleWedgeLayout` does not support fan-out. A goal for the future could be to support all settings for all layouts.

Both `TreeNode` and `RenderEngine` need further clean-up, since both still contain much tree-layout specific code and member variables. Furthermore, `RenderEngine` and its subclasses might benefit from unification to avoid duplicated code. Then it would be clear whether `RenderEngine` needs a different structure, for example if it should use the state pattern instead of subclasses [Gamma et al., 1994, pages 305–313].

Chapter 6

Simple Knowledge Organization System (SKOS)

This chapter introduces the Simple Knowledge Organization System (SKOS), a recent file format which can be used to specify hierarchies.

6.1 Overview

There are many different knowledge organisation systems (KOS) around, such as those related to library science or other KOS like taxonomies and thesauri. SKOS extracts aspects common in all these KOS to create a data model which makes both sharing and connecting KOS on the web easier.

SKOS refers to the W3C recommendation edited by Miles and Bechhofer [2009] (the SKOS Reference). All aspects of SKOS mentioned in this chapter were retrieved from the aforementioned SKOS Reference, the SKOS Primer [Isaac and Summers, 2009] which is a working group note on SKOS, and from separately quoted sources. Whenever a SKOS vocabulary is used, it will be written in the form `skos:Concept` where the first part skos refers to the used namespace `http://www.w3.org/2004/02/skos/core#` and the second, Concept, refers to the SKOS vocabulary. Without this abbreviation, the Uniform Resource Identifier (URI) for `skos:Concept` must be written in the long form `http://www.w3.org/2004/02/skos/core#Concept`.

Resources in a KOS are represented by a set of concept schemes (`skos:ConceptScheme`) and concepts (`skos:Concept`) which are placed in a hierarchy or a network of some sort. SKOS provides a vocabulary for the following cases:

- lexical labels
- notations
- documentation properties
- semantic relations
- concept collections
- mapping properties

Only a subset of these will be discussed here, for more information consult the SKOS Reference [Miles and Bechhofer, 2009].

SKOS is based on the Resource Description Framework (RDF) [Brickley and Guha, 2014a]. Thus, RDF triples are used when describing SKOS resources and their relations. Users are not limited to the

vocabulary provided by SKOS, but can use additional vocabularies like Dublin Core [DCMI, 2012a] or custom vocabularies. This enables the retro-fitting existing KOS with SKOS easily, provided that the KOS can be represented with RDF. One of the main goals of SKOS is to make this process very easy.

The following two examples highlight the simplicity and flexibility of SKOS. In the first example in Listing 6.1, an excerpt of the ACM Computing Classification System [ACM, 2012] created with SKOS is shown. The structure of the document is clearly focused around SKOS. `skos:Concept` and `skos:ConceptScheme` XML elements are used, while just few other elements like `dc:date` and `dc:title` from the Dublin Core Vocabulary are used. Another example, unrelated to computer science, can be seen in Listing 6.2. Of interest is the completely different structure, compared with the ACM Computing Classification. Instead of using `skos:ConceptScheme` XML elements directly, RDF triplets are used to declare that, for example, the RDF description for lithogenetic units is also a `skos:ConceptScheme`. The same method is used for `skos:Concept`.

6.2 Vocabulary

The central piece of SKOS is `skos:Concept` whose definition is kept rather abstract on purpose, so as not to limit uses of SKOS. The SKOS Primer [Isaac and Summers, 2009] refers to concepts as "units of thought which underlie many knowledge organization systems", like an idea, a category of an object or a notion. `skos:ConceptScheme` can be thought of a group of related concepts, like a thesauri or a classification system as was demonstrated in the previous example. The relation can be imagined as a hierarchy or network with a `skos:ConceptScheme` on top and multiple `skos:Concept` below. What form the relation has depends on its participants. Concept schemes are only related to concepts, not to each other, while concepts are related to each other and can also be directly related to concept schemes.

Instances of `skos:Concept` can be grouped into a given `skos:ConceptScheme` using `skos:inScheme` as described in Listing 6.3. Doing that for any concept can be quite burdensome and besides it is not possible to deduce a hierarchy. Nonetheless an advantage of `skos:inScheme` is that it applies to RDF resources, not only to `skos:Concept`, and thus can be used to link other resources to a `skos:ConceptScheme`.

Easier-to-use in this regard are `skos:hasTopConcept` and its reverse `skos:topConceptOf`. `skos:hasTopConcept` is a property in `skos:ConceptScheme` to connect a concept scheme to children of it, which in turn could also have a hierarchy of concepts below them. The reverse of that is `skos:topConceptOf` which is a property of a `skos:Concept` marking it as direct child of the specified `skos:ConceptScheme`. Moreover, a concept can be part of multiple concept schemes.

So far, only relations between a concept and its concept scheme were discussed. SKOS also provides vocabulary to represent semantic relations between concepts themselves. Probably the most important in this regard are `skos:narrower`, `skos:broader`, and `skos:related`. With narrower and broader it is possible to build a hierarchy, while related acts as an association between two concepts which carries no hierarchical meaning. A common usage would be to mark a concept as broader in another concept if the first is more general than the second one. `skos:narrower` works the opposite way.

Concepts can be labelled using SKOS vocabulary. All labels are lexical labels and are represented as strings. Most of these labels are meant to be displayed to users. As a result, it is also possible, but optional, to specify the language the label is in. This allows supporting multiple languages in a KOS with just one SKOS file, as demonstrated in Listing 6.2. The SKOS label types include:

- `skos:prefLabel`, the preferred label, is the common label in use for a resource. For example, "Human" in a hierarchy of species on earth. A resource can have at most one preferred label.
- `skos:altLabel` is an alternative label of a resource, which should obviously be different from the text used for `skos:prefLabel`. A common usage for `skos:altLabel` is to define synonyms of the

```
<skos:ConceptScheme rdf:about="http://totem.semedica.com/taxonomy/The ACM
    Computing Classification System (CCS)">
  <dc:title>The ACM Computing Classification System (CCS)</dc:title>
  <dc:date>2012</dc:date>
  <skos:hasTopConcept rdf:resource="#10002944"/>
  <skos:hasTopConcept rdf:resource="#10002950"/>
  <skos:hasTopConcept rdf:resource="#10002951"/>
  <skos:hasTopConcept rdf:resource="#10002978"/>
  <skos:hasTopConcept rdf:resource="#10003033"/>
  <skos:hasTopConcept rdf:resource="#10003120"/>
  <skos:hasTopConcept rdf:resource="#10003456"/>
  <skos:hasTopConcept rdf:resource="#10003752"/>
  <skos:hasTopConcept rdf:resource="#10010147"/>
  <skos:hasTopConcept rdf:resource="#10010405"/>
  <skos:hasTopConcept rdf:resource="#10010520"/>
  <skos:hasTopConcept rdf:resource="#10010583"/>
  <skos:hasTopConcept rdf:resource="#10011007"/>
  <skos:hasTopConcept rdf:resource="#10011641"/>
</skos:ConceptScheme>
<skos:Concept rdf:about="#10011007" xml:lang="en">
  <skos:prefLabel xml:lang="en">Software and its engineering</skos:
     prefLabel>
  <skos:altLabel xml:lang="en">software</skos:altLabel>
  <skos:altLabel xml:lang="en">software engineering</skos:altLabel>
  <skos:related rdf:resource="#10003022"/>
  <skos:related rdf:resource="#10003463"/>
  <skos:inScheme rdf:resource="http://totem.semedica.com/taxonomy/The ACM
      Computing Classification System (CCS)"/>
  <skos:topConceptOf rdf:resource="http://totem.semedica.com/taxonomy/The
      ACM Computing Classification System (CCS)"/>
  <skos:narrower rdf:resource="#10010940"/>
  <skos:narrower rdf:resource="#10011006"/>
  <skos:narrower rdf:resource="#10011074"/>
</skos:Concept>
<skos:Concept rdf:about="#10011074" xml:lang="en">
  <skos:prefLabel xml:lang="en">Software creation and management</skos:
     prefLabel>
  <skos:altLabel xml:lang="en">software creation</skos:altLabel>
  <skos:related rdf:resource="#10003503"/>
  <skos:inScheme rdf:resource="http://totem.semedica.com/taxonomy/The ACM
      Computing Classification System (CCS)"/>
  <skos:broader rdf:resource="#10011007"/>
  <skos:narrower rdf:resource="#10011075"/>
  <skos:narrower rdf:resource="#10011081"/>
  <skos:narrower rdf:resource="#10011092"/>
  <skos:narrower rdf:resource="#10011099"/>
  <skos:narrower rdf:resource="#10011111"/>
  <skos:narrower rdf:resource="#10011134"/>
</skos:Concept>
```

Listing 6.1: An excerpt from the ACM Computing Classification System created with SKOS [ACM, 2012], which mostly uses just SKOS vocabulary, with two elements from the Dublin Core vocabulary.

```
<rdf:Description rdf:about="http://resource.geolba.ac.at/GeologicUnit/1">
  <rdf:type rdf:resource="http://www.w3.org/2004/02/skos/core#
      ConceptScheme"/>
  <rdf:type rdf:resource="http://www.w3.org/2000/01/rdf-schema#Resource
      "/>
  <skos:hasTopConcept rdf:resource="http://resource.geolba.ac.at/
      GeologicUnit/354"/>
  <dcterms:publisher>Geological Survey of Austria</dcterms:publisher>
  <dcterms:title xml:lang="en">Lithogenetic Units</dcterms:title>
  <skos:hasTopConcept rdf:resource="http://resource.geolba.ac.at/
      GeologicUnit/358"/>
  <skos:hasTopConcept rdf:resource="http://resource.geolba.ac.at/
      GeologicUnit/346"/>
</rdf:Description>
<rdf:Description rdf:about="http://resource.geolba.ac.at/GeologicUnit
    /354">
  <rdf:type rdf:resource="http://www.w3.org/2004/02/skos/core#Concept"/>
  <rdf:type rdf:resource="http://www.w3.org/2000/01/rdf-schema#Resource
      "/>
  <skos:narrowerTransitive rdf:resource="http://resource.geolba.ac.at/
      GeologicUnit/333"/>
  <skos:narrower rdf:resource="http://resource.geolba.ac.at/GeologicUnit
      /333"/>
  <skos:narrowerTransitive rdf:resource="http://resource.geolba.ac.at/
      GeologicUnit/335"/>
  <skos:narrower rdf:resource="http://resource.geolba.ac.at/GeologicUnit
      /335"/>
  <skos:prefLabel xml:lang="de">Hangablagerung</skos:prefLabel>
  <skos:prefLabel xml:lang="en">Slope sediment</skos:prefLabel>
  <skos:topConceptOf rdf:resource="http://resource.geolba.ac.at/
      GeologicUnit/1"/>
</rdf:Description>
```

Listing 6.2: Simplified excerpt from a thesaurus in SKOS for lithogenetic units [GBA, 2014].

```
<skos:ConceptScheme rdf:about="http://example.com/"/>
<skos:Concept rdf:about="#1">
  <skos:inScheme rdf:resource="http://example.com/"/>
</skos:Concept>
```

Listing 6.3: Example of the usage of skos:inScheme.

preferred label. For the above example, that could be “Homo sapiens” as the Latin name of the species used in English. There can be multiple alternative labels in the same language, like having both “Homo sapiens” and “Home sapiens sapiens”. Alternative labels can be present without any preferred label.

- `skos:hiddenLabel` is not supposed to be displayed to users, contrasting it from the other available labels. Instead, it could be used, say for indexing operations by applications.

Further information can be related to resources using documentary notes. There is no restriction to just plain text, for example, even images and videos could be related to a resource. The following documentary notes are available:

- `skos:note` can be used for general information, it is the base for all other documentary notes available.
- `skos:scopeNote` gives information on the scope of the resource. For example, in a software-related KOS, this could give information in which situation a class should be used.
- `skos:definition`
- `skos:example`
- `skos:historyNote` gives information of the history of the resource. In a software-related KOS, for example, `skos:historyNote` could be used to describe the context that lead to using the term “bug” for software errors.
- `skos:editorialNote` is not meant to be displayed to users of a KOS. Instead, it is focused on the people maintaining the KOS. For instance, an editorial note could be used if specific parts of a resource need more work or refinement.
- `skos:changeNote` is also focused on the people maintaining the KOS, making it easier for them to follow modifications to the resources.

Chapter 7

Implementing SKOS for HVS

HVS already supported hierarchies based on local file systems and those stored in TreeML format. For more information on TreeML consult TreeML [2006] and Fekete and Plaisant [2003]. The goal of this implementation was to add SKOS support to HVS, so that KOSes realised with SKOS could be navigated inside HVS using the available visualisations. HVS uses a model-view-controller (MVC) architecture, where input modules provide data of the currently investigated hierarchy. This decouples the rest of the code from the input modules, adding flexibility. There are only a few restrictions on the internals of an input module, giving developers freedom for the implementation. The details of creating a new input module are skipped here, an overview is given by Putz [2005, pages 39–41].

The `SkosDataModel` class represents a SKOS KOS in the form of a hierarchical model. The hierarchy is built from top to bottom:

1. Read in all concept schemes S. If there is just one concept scheme s then treat it as root of the hierarchy. Otherwise create a fake root f and add each concept scheme s as a direct child to the fake root f.

2. Read in each concept c marked with `skos:hasTopConcept` in concept scheme s and add them as child to s.

3. When reading in a concept c, also read in all concepts N marked with `skos:narrower` inside of c and add these concepts to c as children. Proceed as long as there are no more concepts to read in.

This very easy approach is a depth-first-search [Sedgewick and Wayne, 2011, pages 530–534]. Graphs are also supported, by implicitly converting them to a hierarchy with the aforementioned algorithm. Since `skos:related` has no hierarchical information it is not used to create the hierarchy.

None of the SKOS labels are compulsory. Therefore, a valid SKOS KOS might not have any SKOS labels present. To determine a name for a node, which might be displayed to users, the following properties are considered in order:

1. `skos:prefLabel`

2. `skos:altLabel`

3. Dublin Core Terms `dcterms:title` [DCMI, 2012c].

4. Dublin Core `dc:title` [DCMI, 2012b], which is an underspecified predecessor of `dcterms:title` DCMI [2013].

5. `rdfs:label` [Brickley and Guha, 2014b].

Thus, if any of these properties is present, the node will have a name. If a label is present in multiple languages, then English is preferred.

Whenever a resource is read in, metadata is collected. This metadata is displayed when hovering over a node (if tool tips are activated) and in the Properties Panel, as shown at the bottom of Figure 7.1. The Properties Panel displays any available metadata for the resources. As soon as any resource has the `skos:related` property, for example, the Related column is displayed for all resources. The following metadata fields are available:

- Metadata for which only plain text is supported:
  - `skos:altLabel`
  - `skos:note`
  - `skos:defintion`
  - `skos:scopeNote`
  - `skos:historyNote`
  - `skos:example`
  - `dcterms:description`
- Metadata with date and time support:
  - `dcterms:modified`
- Metadata containing links to other resources:
  - `skos:related`
  - `skos:broader`
  - `skos:broaderTransitive`
  - `skos:narrower`
  - `skos:narrowerTransitive`

A special feature of the implementation is that whenever a cell is double-clicked in the Properties Piew, a search is created automatically for the metadata under the cursor. Thus, if the Modified cell is double-clicked, all resources are searched for which were modified at the same time. Similarly, whenever a cell containing links to other resources is double-clicked, then these resources are searched for. For example, in Figure 7.1, the Related cell of node Scarp was double-clicked, leading to the highlighting of all the nodes listed there, which are those linked to Scarp via the `skos:related` property. This assists in understanding relations between nodes in the hierarchy.

HVS differentiates between highlighting and selecting nodes. Selecting nodes does not change their highlight status, which allows to select nodes while still having the search result present. Moreover, the node currently selected in the separate search result window is highlighted in red [Putz, 2005, pages 44–45].

7.1 Library Considerations

In software development, code reuse is important. As a result, when adding support for SKOS to HVS, one goal was to reuse as much existing code as possible. This reduces both initial programming and maintenance effort for HVS.

The SKOS API [Jupp, Bechhofer, and Stevens, 2009; Jupp, 2013] is an API for SKOS written in JAVA, which initially appeared to be suitable, since it is directly tailored to SKOS. Another advantage of

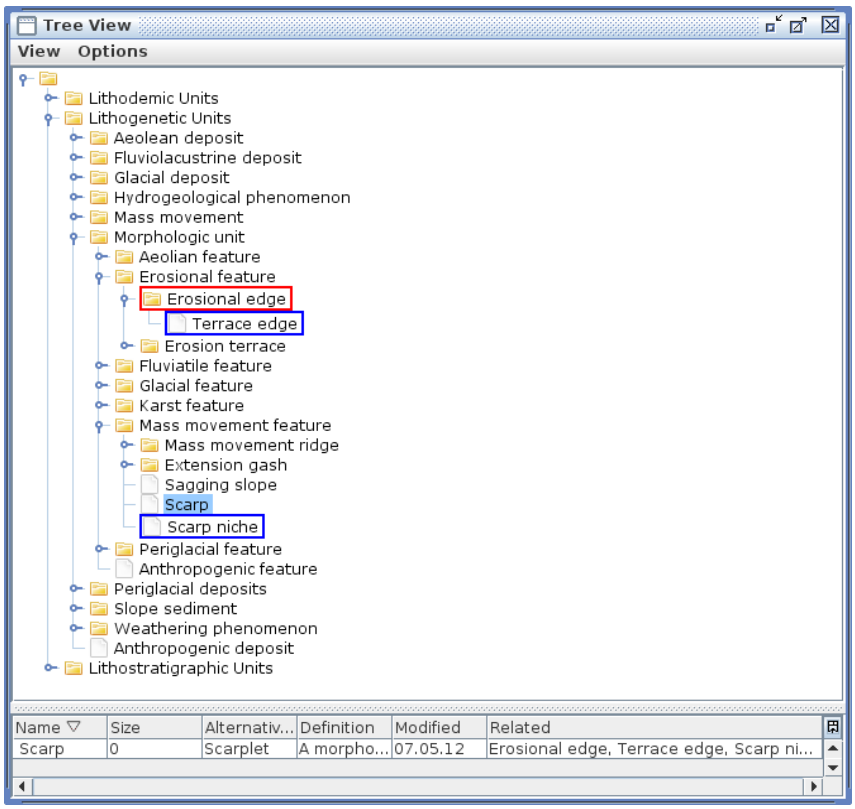


Figure 7.1: SKOS metadata retrieved from a node of a hierarchy [GBA, 2014] displayed in HVS. The Properties Panel at the bottom displays metadata for selected nodes. Nodes related to node Scarp via `skos:related` were highlighted after double-clicking on the Related cell of Scarp. [Screenshot taken by the author of this thesis.]

the SKOS API is the reuse of the OWL API [Horridge and Bechhofer, 2011; OWL, 2014], which reduces the maintenance burden for the SKOS API developers and thus makes it easier form them to maintain the SKOS API itself.

However, after initial trials, problems were discovered: Many SKOS files, like ACM [2012], were not parsed correctly, without any hint of what could have gone wrong. Little error handling is available. Errors are often hidden deep in the code and are not accessible. After contacting the creator of the SKOS API, it turned out that no URIs with white space are supported by the OWL API implementation and as a result these are not supported in the SKOS API. Fixing the related SKOS file still did not make parsing the hierarchy possible. It turned out that both the SKOS API and the OWL API have an unnecessarily complex architecture, which makes both using and debugging the library very hard.

For example, every class in the SKOS API implements an interface for no apparent reason [Feathers, 2004, page 134]. That means that the whole file structure is duplicated: one file for the interface and another file for the implementation with an Impl postfix in the name. Using interfaces is a common technique to ensure loose coupling between the components of a program. The technique is also used to break dependencies and to make code more testable, which is, amongst others, described by Feathers [2004, pages 356–368]. Yet, there are neither unit tests for the SKOS API, nor are there multiple implementations. Moreover, despite the interface usage, writing tests for the SKOS API is complex, as the abstraction is broken in many areas: Interface instances are often cast to the implementation classes inside of the implementation, which complicates both test writing and enhancing the API with custom classes. Many dependencies are obfuscated as a result. This could lead to runtime errors when used incorrectly. What incorrect API usage means is not clear though, due to the lacking documentation. For better information, users would have to look at the implementation themselves. To make matters worse the API itself is hard-to-use: Most API calls expect some parameters and then an instance of `SKOSDataset` which does the actual work of the API call. The SKOS API depends heavily on the abstract factory pattern [Gamma et al., 1994, pages 87–95] with its `SKOSDataFactory` interface, thus there is no reason why class instances created by the factory do not receive a `SKOSDataset` instance automatically.

All these drawbacks lead to the decision to use a different library, the RDF library by Apache which is part of the Jena framework [Apache, 2014b]. Not only is the Jena API better designed than those of both the SKOS API and the OWL API, but also parsing SKOS files worked successfully. Implementing wrapper classes [Gamma et al., 1994, pages 139–150] on top of the Jena API was easy. With the help of these wrapper classes accessing SKOS related information was simplified.

7.2 Outlook and Further Work

SKOS support could be made more flexible. For instance, at the moment only single SKOS files are supported. An interesting goal would be to support SKOSes distributed over multiple servers. This would need the implementation of the advanced parts of SKOS [Isaac and Summers, 2009]. Moreover, HVS could be extended to optionally draw edges for associatively related nodes.

Chapter 8

Selected Details of the Implementation

The HVS architecture was refined and refactored during development using proven techniques, described for example in Fowler [1999] and Feathers [2004]. The focus was on the actual purpose of HVS: visualising hierarchies. Parts which made this harder, like the plug-in architecture, were modified or removed. Moreover, duplication was reduced in the code base and existing visualisations were refined and extended with new features. Some of these changes will be described in the following sections.

8.1 No Plug-Ins

Originally, the HVS architecture was heavily plug-in-based. The plug-ins consisted of byte code in the form of *.class*-files and XML files. The XML files described the structure of the plug-ins: their name and their purpose. New visualisations as well as data sources could be added to an existing HVS installation by copying the binary code to the correct directories and by adding plug-in definitions in XML files.

This flexibility is hardly needed though since HVS is in general distributed as a whole and no single updates of files are shipped. Moreover, the plug-in architecture made developing HVS harder:

- Opening the project in an IDE was not enough.
- Adding new visualisations and data sources was complicated.
- Releases in self-sufficient jar-files were impossible.

As a result, the whole plug-in architecture was removed and instead one jar-file can be shipped now. The Ant [Apache, 2014a] build file was heavily simplified during that refactoring process and will automatically detect new libraries, new source directories, and new resources like icons. If there is ever the need for plug-ins again, more flexible technologies are available today than were available when HVS was created.

8.2 Improved Settings Handling

In HVS, most of the available user settings were not kept persistent, thus after closing HVS these settings were lost. Some settings were kept during sessions and were stored in the form of property files or XML files parsed with SAX [Megginson, 2004]. The handling of the properties was very complicated and resulted in much code just to support one setting which consisted of a default value, a graphical way to change it and code reacting to these changes. Moreover, the parsing of XML files was also very complicated and needed much code in the implementation. All of these parts were hand-coded making the addition of new settings at least as complicated as the addition of previous ones. A new settings

system was introduced which focuses on ease-of-use for both users and developers. This new system is described in the following subsections.

8.2.1 Structure

A class hierarchy was added which provides classes for the most common types of settings:

- Boolean settings.
- Range of valid integer values.
- Range of valid floating point values with a given precision.
- Settings for colour values.
- String settings which can take any value of a provided set of strings.

All of these settings have a default value and can be enabled and disabled.

8.2.2 Persistence

Persistence of settings is very important, otherwise users have to repeatedly perform the same configuration tasks whenever they start HVS. Therefore, each of the mentioned settings can be kept persistent easily using the marshalling capabilities of JAXB [JAXB, 2014]. JAXB takes care of both serialising and deserialising. Using JAXB instead of SAX avoids much non-reusable code: For example, the icon configuration had 119 Source Lines of Code (SLOC) determined with SLOCCount [Wheeler, 2004]. Using JAXB, only 19 SLOC are required.

8.2.3 Reacting to Changes

Whenever a setting was modified all the affected parts of HVS had to be informed manually of the change. This was the responsibility of the object modifying the setting. To accomplish this, the object, for example, a settings dialogue box, had to know every user of the setting and then inform them. This resulted in tightly coupling the dialogue box to the places the settings were used and such code coupling is undesirable [Feathers, 2004].

To avoid this, the new settings system supports the observer pattern described in Gamma et al. [1994, pages 293–303]. So-called listeners can be registered on specific setting instances. These listeners are informed once the setting they listen to is modified, enabling them to act on changes, without knowing what the source of change was. For example, there is a setting for line colour. A dialogue box which is used to change the line colour setting only needs the setting object and can then retrieve the current value, set the value, and revert to the default value. Changing the line colour, leads to a redrawing of the view by the notified listeners. A listener itself need not know that it is listening, this connection can be established at a higher level.

8.2.4 User Interaction

Many settings should be configurable by users. Creating setting dialogue boxes was made easy. An example dialogue box can be seen in Figure A.3 on page 63. Each setting type has an associated graphical representation. For instance, boolean settings are represented with a check box, while integer settings can be modified with a slider which also supports the mouse wheel as input device. Sometimes, specific settings are not available or not supported by the current layout. These settings can be disabled which

```
// GraphViewSetting.java
@XmlRootElement
@XmlType
public class GraphViewSetting
{
  public final ColorSetting linecolor = new ColorSetting(Color.GRAY);
}

// DialogRenderSettings.java
public class DialogRenderSettings extends Dialog
{
  public DialogRenderSettings(GraphViewSettings settings, JPanel parent)
  {
    super("Rendering Settings", parent);
    addComponent("Line Colour", settings.linecolor);
    prepare();
  }
}

// HvsView.java
private void registerSettingsListeners()
{
  getGraphViewSettings().linecolor.addListener(new ReRender());
}
```

Listing 8.1: Code example showing the ease of handling settings with the new setting framework. A colour can be set via a dialogue box, which automatically leads to the scene being re-rendered. [Source code written by the author of this thesis.]

automatically greys them out in the dialogue box and forbids interaction with them. All these building blocks are provided and can be reused when creating a new settings dialogue box.

Listing 8.1 shows the code needed to provide a line colour setting. The code is taken from HVS, removing unnecessary surrounding code for clarity. Using listeners leads to better interactivity, since changes are instantly reflected in the view. For example, changing the slider for the font size or changing the line colour instantly leads to an updated view. Thus, any changes are visible before pressing the OK button. Pressing the Cancel button instantly reverts to the situation before the dialogue box was opened, while pressing the Defaults button resets every value to its default setting.

For this example, it is also of interest that `DialogRenderSettings` consisted of 259 SLOC before adapting it to the new framework, while only 17 SLOC are needed afterwards. Moreover, all of these settings are now persistent.

8.2.5 Settings Per View Instance

Most of the settings available in HVS were not stored in the saved sessions. The settings which were saved were not saved per view instance, but instead per view type. When loading previously stored settings, all views of the same type looked alike, even if they were configured differently in the previous user session. This resulted in tedious work to have the loaded session look and behave the way it did when it was saved.

The new design stores the settings per view. This is especially useful when comparing the same visualisation with different settings, for example, comparing the effect of the many different radial tree drawing algorithms using a particular data set.

8.3 Walker Algorithm

The implementation of the Walker algorithm [Walker, 1990] in HVS contained a bug, which lead to overlapping nodes. Both the original Walker algorithm and the better performing version by Buchheim, Jünger, and Leipert [2002] expect nodes to have an ancestor. This ancestor is used when moving a subtree to the right to ensure that nodes are not too close. The determination of the ancestor was wrong in HVS. As a result, subtrees were not moved far enough, leading to the aforementioned overlapping nodes. Figure 8.1 shows a hierarchy visualised on the left with the broken algorithm and on the right with the fixed algorithm. By fixing this issue, all visualisations in HVS relying on the Walker algorithm were also improved, including: the Dendrogram Browser, the InfoLens Browser, the Magic Eye Browser and, of course, the Walker Tree Browser itself.

8.4 Other Changes

Some of the other changes made to HVS in the course of this thesis work include:

- Maven [Apache, 2014c] was investigated as a build system, but was reverted back to Ant [Apache, 2014a]. The latter works better without an internet connection. Furthermore, the build files were adapted to create a self-sufficient jar file which can be distributed without any further dependencies.
- Consistent mouse interaction was implemented for all views:
  - Pressing left mouse button and dragging pans the view.
  - Pressing the shift key and then left-clicking and dragging creates a selection box.
  - Left- or right-clicking on an empty space deselects all nodes.

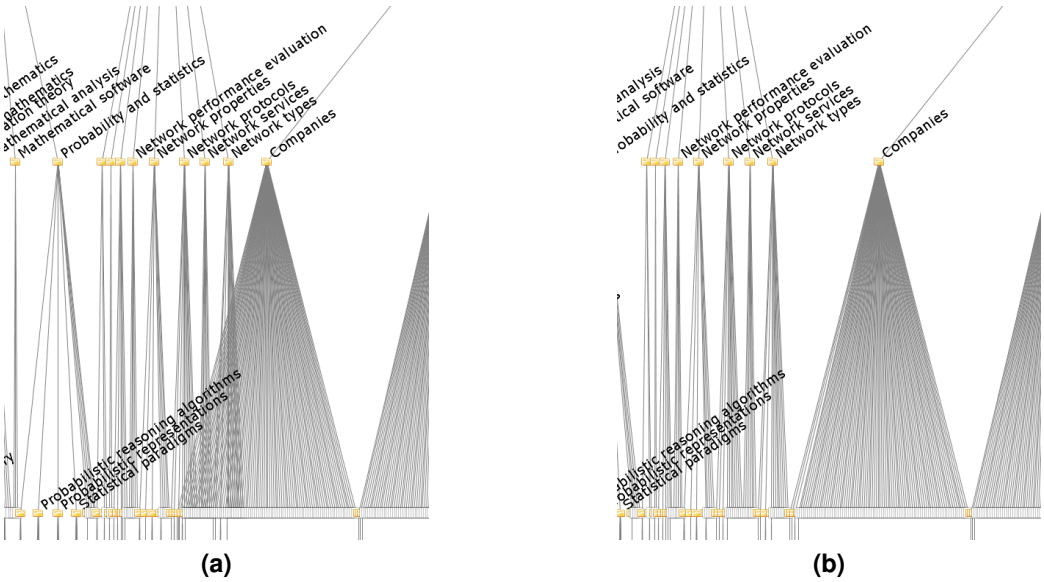


Figure 8.1: Both screenshots show a visualisation of the ACM Computing Classification System [ACM, 2012] using the Walker Tree Browser. The Companies node is zoomed in. In the original HVS implementation shown in (a), there are overlapping nodes. With the algorithm fixed, in (b), there are no overlapping nodes. [Screenshots taken by the author of this thesis.]

  - Right-clicking on an empty space shows a context menu, but with fewer available options. This is done to inform users that their input was recognised.
  - The context menu works under Linux. Cases where the context menu could not be dismissed anymore were fixed.
  - A node is focused and selected when the context menu is opened on it, which helps to understand which node the actions will be applied on.
  - Zooming in is mapped consistently to scrolling up with the mouse wheel.

- HVS now uses the correct widgets in more cases. Check boxes are used for choices which are independent of each other. Radio boxes are used for dependant choices.
- Performance was improved when using system icons by caching them. They are used by default now.
- HVS now uses the Silk [James, 2006] icon theme, if the system icons are not used.
- Treemap options were moved from the sidebar to a dialogue box. The new settings system is used.
- Treemap supports interaction with the mouse wheel. Scrolling up results in moving one level down at the mouse cursor location, while scrolling down results in moving one level up.
- When choosing a data source, the open dialogue will automatically point to the previously opened location.
- Automatic line breaks have been added to tool tips for very long metadata.

Many refactorings were done to HVS, as illustrated by a small example. The Walker tree browser and the Dendrogram browser consisted of much duplicated code, caused by the latter being a copy of the former. Code common between these two browsers was moved into newly created parent classes, while settings-related code was simplified. Table 8.1 shows the results: The code base was reduced by nearly 50%. The new settings framework avoided much duplicated code and also made the settings persistent.

Directory	before [SLOC]	after [SLOC]
basictree	1194	212
dendrogram	1525	304
scrollable	0	853
	2719	1369

Table 8.1: Refactoring the Walker tree browser (basictree) and the Dendrogram browser reduced the code base by roughly 50%. Most of the code is shared now in the scrollable directory. Much code duplication was also avoided by using the new settings framework.

Chapter 9

Concluding Remarks

This thesis described the extension of the Hierarchical Visualization System (HVS) with a radial tree visualisation and SKOS import functionality. HVS was not only extended during the course of this thesis, but also modernised and refactored.

The first chapter gave a theoretical overview of information visualisation. Different forms of visualisation were described and consideration was given to human capabilities to create clear and easily understandable visualisation systems.

Chapter 3 delved into hierarchy visualisation. A history of trees used in visualisation was presented, which also highlighted the influence trees have had on humans over the centuries. Trees, both in their abstract and also in their botanical form, are used to illustrate the concept of hierarchy. Tree visualisations were categorised into node-link trees and space-filling trees, giving examples for each. Requirements for tree visualisations were also discussed. To conclude the chapter, examples were given for practical uses of hierarchy visualisation, emphasising the benefit of including multiple, synchronised visualisations in a single program.

The main contribution of this thesis is contained in Chapters 4 and 5. Chapter 4 described radial tree visualisations by categorising them into fixed-wedge layouts and flexible-wedge layouts. During the course of this chapter, aspects of radial tree layouts were discussed and illustrated with figures which lead to the creation and adaption of algorithms. It was investigated how to define the annulus wedge which restricts the space nodes of a subtree are allowed to use. Chapter 5 described the implementation of radial tree visualisations in HVS. Aside from the different layout algorithms, user interactions were also implemented: The radial tree browser supports zooming and panning and reacts to changes in focused nodes. Moreover, subtrees can be fanned out and navigation to nodes can be done in an animated fashion using a focus+context technique. The radial tree browser provides different configuration options per browser window, which can be changed at any time and saved for later sessions.

Chapter 6 discussed the idea behind the Simple Knowledge Organization System (SKOS) and its structure, with examples of hierarchies stored in SKOS. Adding import functionality to HVS for SKOS was described in Chapter 7. The SKOS implementation provides not only import functionality but also includes means to display the semantic information associated with nodes and to highlight related nodes.

Finally, modernisations and refactorings of HVS were described in Chapter 8. Amongst other improvements, a framework was introduced which makes adding user-configurable settings easy. Every setting realised with this framework has a consistent look and behaviour, supports cancelling changes, reverting to the defaults, and can also be kept persistent across sessions.

Appendix A

User Guide

This user guide describes the usage of newly added features to HVS. General user guides for HVS already exist within Putz [2005, pages 95–109] and Nussbaumer [2005, pages 91–103]. These are highly recommended.

A.1 Installing and Starting HVS

The installation of HVS changed from Putz [2005, page 95] in the regard that the different browsers are no longer plug-ins. Therefore for installation, it is enough to extract the received zip file. HVS can then be started by double-clicking `hvs.bat` on Windows and using `hvs.sh` on Unix systems. Alternatively, it is also possible to distribute a single `hvs.jar` file which only needs to be double-clicked to start HVS.

A.2 Radial Tree Browser

The radial tree browser is a visualisation which uses a radial tree layout to present the selected hierarchy. Before it can be used, a data source has to be chosen from *File* → *Data Source*, which is described in more detail in Putz [2005, page 96]. Afterwards, "Radial Tree" has to be selected in *File* → *Visualisation*. This opens a sub-window like the one shown in Figure A.1 containing a radial tree visualisation of the hierarchy.

A.2.1 Mouse Interaction

The mouse is used for interaction with the radial tree browser. The following actions are available.

- **Left Button**
  - **Single-Click:** Selects the node under the cursor. or deselects all nodes if no node is under the cursor.
  - **Control + Single-Click:** Adds the clicked node to the already selected nodes. If no node is under the cursor the current selection is left unchanged.
  - **Double-Click:** Focuses the node under the cursor, which leads to a focus event for every window. See Section A.2.2 for the possible reactions to a focus event.
  - **Drag:** Pans the view.
  - **Shift + Drag:** Creates a selection box to select all nodes inside that box.

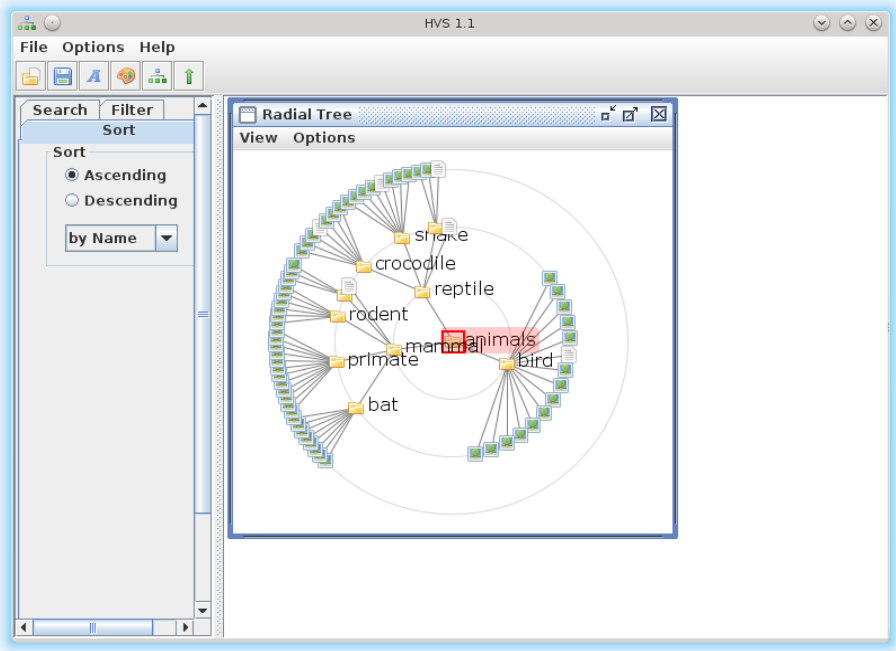


Figure A.1: HVS after a data source was chosen and the radial tree visualisation was selected. [Screenshot taken by the author of this thesis.]

- **Middle Button:** A single-click on a node opens the node with an associated handler program. For example, if the node corresponds to an image on the hard drive, an image viewer is opened pointing to that image. Not every node has an associated program.
- **Right Button:** A single-click opens the context menu. The clicked node also receives focus.
- **Mouse Wheel**
  - **Scroll:** Scrolling up zooms in, while scrolling down zooms out.
  - **Control + Scroll:** Scrolling up increase the fan-out level of the focused node, while scrolling down decreases the fan-out level, thereby fanning in.

A.2.2 Options Menu

Most of the options in the Options menu in Figure A.2 are self-explanatory and are already described in Nussbaumer [2005, pages 100–101]. Only the layout settings of Figure A.3 are described here. All of the following settings are applied immediately. Thus, clicking the OK button only hides the dialogue box. This is especially useful for options which use sliders. Pressing the Cancel button restores the settings to their values before the dialogue box was shown, while Defaults resets the settings to their factory default values.

- **Show Rings:** When ticked, the concentric rings are shown, otherwise hidden.
- **Layout:** Available radial tree layouts:
  - **Flexible Wedges:** Each node receives an annulus wedge of the circle within which it is placed. A child node's wedge can be enlarged by empty space between them and their direct neighbours. For example, if on a particular level, the following nodes exist: A, B and C, where only A and C have children, these children can utilise the space unused by B.

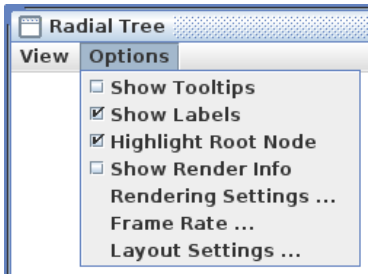


Figure A.2: The available options of the radial tree browser. Most of the menu entries are explained in Nussbaumer [2005, pages 100–101]. [Screenshot taken by the author of this thesis.]

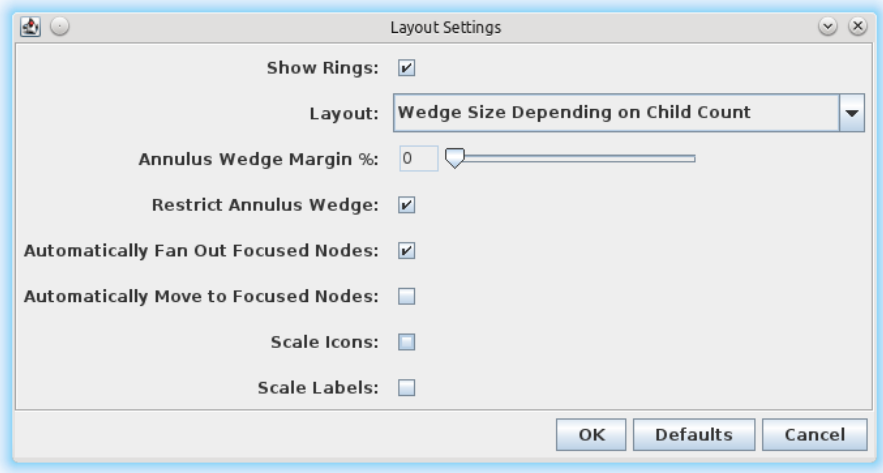


Figure A.3: Layout settings of the radial tree browser. [Screenshot taken by the author of this thesis.]

  - **Same Wedge Size per Sibling:** Every sibling has the same amount of space.
  - **Same Wedge Size per Sibling Except First Level:** Every sibling has the same amount of space, only nodes on the first level receive space based on the number of children they have.
  - **Walker Layout:** Uses the Walker tree layout adapted to radial trees.
  - **Wedge Size Depending on Child Count:** The default layout. Nodes with many children receive more angular width to place their children in.

- **Annulus Wedge Margin:** The percentage of the annulus wedge used as a margin between neighbouring annulus wedges. The setting can also be modified by placing the mouse cursor over the slider and using the mouse wheel.

- **Restrict Annulus Wedge:** When activated, the annulus wedge is restricted to ensure there are no edge crossings. Otherwise, space is used more efficiently at the cost of possible edge crossings.

- **Automatically Fan Out Focused Nodes:** When a focus event is received, the focused node will be fanned out using an animated transition. Thus, it and its children will receive more space. If the node was a leaf node, then its siblings will also receive more space. Any already fanned out nodes will be fanned in first.

- **Automatically Move to Focused Nodes:** Will pan to the focused node when a focus event is received, unless the node is already visible. The animated transition uses zooming to provide more context and to finish the operation faster. When both this and the previous setting are activated, fan out will be done for currently visible nodes, while the moving will be done for currently invisible nodes.

- **Scale Icons:** If ticked, the icon size depends on the density of the concentric circles: The denser these circles are, the smaller the icons are drawn. At some point, icons will no longer be drawn. Icon size is related to font size, they cannot become larger than the pre-configured font size. Information on how to change the font size is given in Putz [2005, pages 97 and 100].

- **Scale Labels:** Behaves like the previous option, only that labels are affected instead of icons.

A.3 SKOS

The SKOS data source can be selected by choosing SKOS from *File* → *Data Source*. The user has to choose a local SKOS file in the subsequent dialogue box.

There are two different ways to access semantic information associated with resources in the chosen SKOS file:

- Activating tooltips for a visualisation by ticking *Options* → *Show Tooltips* and then hovering over nodes.

- Displaying the properties in the Properties Panel of a visualisation via *View* → *Show Properties* and then selecting nodes.

Using *Show Properties* allows further interactivity: Figure A.4 shows two nodes selected in the tree view and the metadata of these nodes displayed in the Properties Panel below. Double-clicking on any cell in that table initiates a search for other nodes having the same value in that cell. Furthermore, there are some special cells containing links to other nodes, like the *Related* or the *Broader* field. If such a special cell is double-clicked, the nodes listed there are highlighted in the visualisation. For example, in Figure A.5 the *Related* cell of "Scarp" was double-clicked. As a result, the nodes "Erosional Edge", "Terrace Edge", and "Scarp niche" were then highlighted in the visualisation. Search results

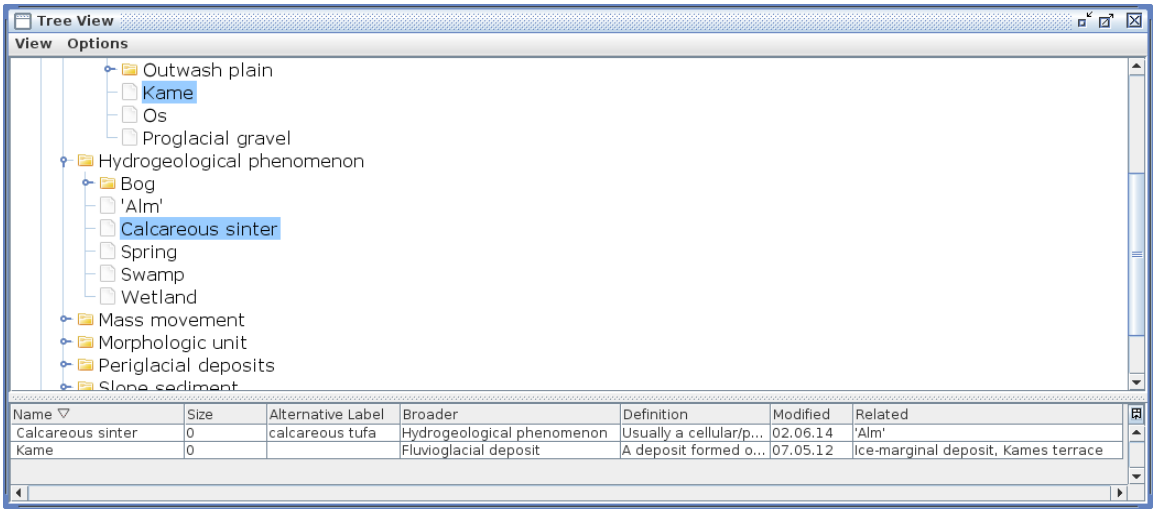


Figure A.4: The Properties Panel at the bottom of the visualisation displays the metadata of selected nodes in a table. Double-clicking a cell in the table initiates a search for all nodes having the same property. [Screenshot taken by the author of this thesis.]

are visualised by highlighting nodes with an unfilled rectangle, which in HVS is different from selected nodes. A node can be both selected and highlighted at the same time, allowing to select nodes while the search results are still visualised. The node currently selected in the separate search result window is highlighted in red [Putz, 2005, pages 44–45]. The search result window can be displayed by checking the *Search Result List* option in the *Search* tab of the side panel.

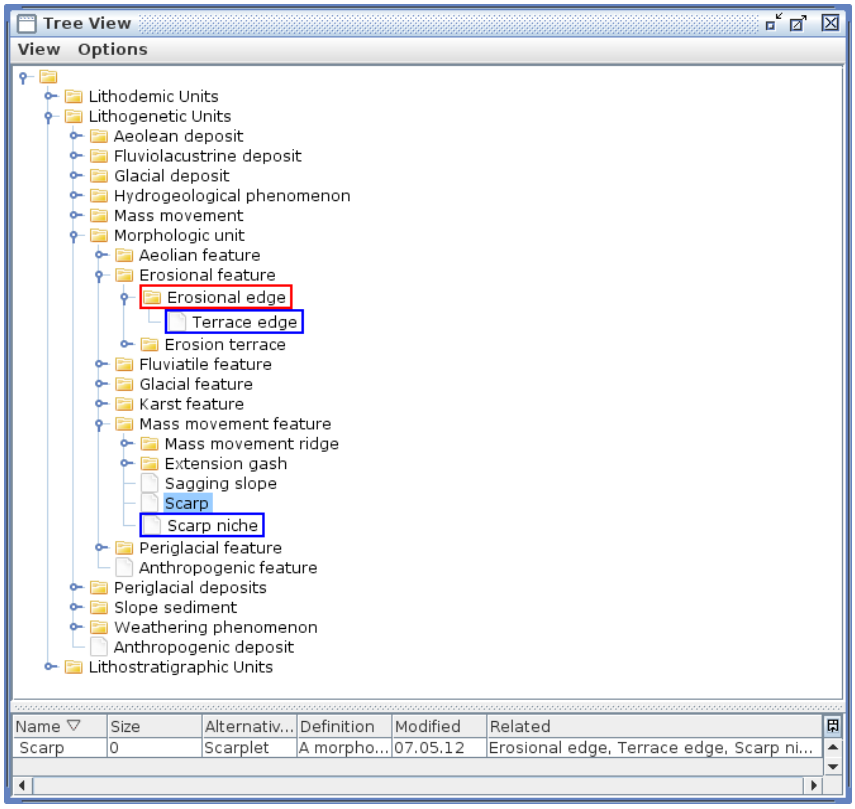


Figure A.5: The *Related* cell of "Scarp" was double-clicked. Since *Related* is a special cell whose content refers to other nodes, all nodes listed in the cell (those related to node "Scarp") were then highlighted in the visualisation in blue. The cell highlighted in red refers to the node that is currently selected in the search results window, which can be displayed by checking the *Search Result List* option in the *Search* tab of the side panel. [Screenshot taken by the author of this thesis.]

Bibliography

ACM [2012]. *The 2012 ACM Computing Classification System*. Association for Computing Machinery. 2012. http://acm.org/about/class/2012 (cited on pages 40, 44, 45, 52, 57).

Andrews, Keith [2014]. *Writing a Thesis: Guidelines for Writing a Master's Thesis in Computer Science*. Graz University of Technology, Austria. The author of this thesis was provided with a more recent version than is available online. Jan. 23, 2014. http://ftp.iicm.edu/pub/keith/thesis (cited on page xi).

Andrews, Keith [2015]. *Information Visualisation: Course Notes*. Mar. 11, 2015. http://courses.iicm.tugraz.at/ivis/ivis.pdf (cited on pages 3, 7).

Andrews, Keith and Werner Putz [2005]. *HVS: A Framework for Visualising Hierarchies*. Software Demo, Graph Drawing 2005 (GD2005), Limerick, Ireland. Sept. 12, 2005. http://ftp.iicm.tugraz.at/pub/papers/andrews-gd2005-demo-hvs.pdf (cited on page 1).

Andrews, Keith, Werner Putz, and Alexander Nussbaumer [2007]. "The Hierarchical Visualisation System (HVS)". In: *Proc. 11th International Conference on Information Visualisation (IV'07)*. (Zurich, Switzerland). IEEE Computer Society Press, July 2, 2007, pages 257–262. doi:10.1109/IV.2007.112. http://ftp.iicm.tugraz.at/pub/papers/andrews-iv2007-hvs.pdf (cited on page 35).

Apache [2014a]. *Apache Ant*. The Apache Software Foundation. May 23, 2014. http://ant.apache.org/ (cited on pages 53, 56).

Apache [2014b]. *Apache Jena*. The Apache Software Foundation. 2014. http://jena.apache.org/ (cited on page 52).

Apache [2014c]. *Apache Maven*. The Apache Software Foundation. Aug. 18, 2014. http://maven.apache.org/ (cited on page 56).

Battista, Giuseppe Di et al. [1999]. *Graph Drawning: Algorithms for the Visualization of Graphs*. Prentice Hall, 1999. ISBN 0133016153 (cited on pages 17, 18, 25).

Book, Greg and Neeta Keshary [2001]. *Radial Tree Graph Drawing Algorithm for Representing Large Hierarchies*. Dec. 2001. http://gbook.org/projects/RadialTreeGraph.pdf (cited on pages 18, 21, 23, 29).

Brickley, Dan and R.V. Guha, editors [2014a]. *RDF Schema 1.1*. W3C Recommendation. Feb. 25, 2014. http://w3.org/TR/2014/REC-rdf-schema-20140225 (cited on page 43).

Brickley, Dan and R.V. Guha, editors [2014b]. *RDF Schema 1.1: Label Property*. W3C Recommendation. Feb. 25, 2014. http://w3.org/TR/2014/REC-rdf-schema-20140225/#ch_label (cited on page 49).

Buchheim, Christoph, Michael Jünger, and Sebastian Leipert [2002]. "Improving Walker's Algorithm to Run in Linear Time". In: *Proc. 10th Symposium on Graph Drawing (GD 2002)*. Berlin, Heidelberg: Springer, Aug. 2002, pages 344–353. ISBN 3540001581. doi:10.1007/3-540-36151-0_32. http://e-archive.informatik.uni-koeln.de/431/1/zaik2002-431.ps (cited on page 56).

Burch, Michael et al. [2011]. "Evaluation of Traditional, Orthogonal, and Radial Tree Diagrams by an Eye Tracking Study". *IEEE Transactions on Visualization and Computer Graphics* 17.12 (Dec. 2011), pages 2440–2448. ISSN 1077-2626. doi:10.1109/TVCG.2011.193. http://www.vis.uni-stuttgart.de/~weiskopf/publications/infovis11_eyetracking.pdf (cited on pages 7, 15).

Carriere, Jeromy and Rick Kazman [1995]. "Interacting with Huge Hierarchies: Beyond Cone Trees". In: *Proc. 1st IEEE Symposium on Information Visualization (InfoVis '95)*. (Atlanta, GA, USA). IEEE, Oct. 1995, pages 74–81. ISBN 0818672013. doi:10.1109/INFVIS.1995.528689 (cited on page 9).

Chiang, David [2012]. *tikz-qtree: Better Trees with TikZ*. Apr. 22, 2012. http://isi.edu/~chiang/software/texmf/tikz-qtree-manual.pdf (cited on pages 7, 8).

Creative Commons [2010]. *Public Domain Mark 1.0*. Oct. 12, 2010. http://creativecommons.org/publicdomain/mark/1.0 (cited on pages xi, 16).

DCMI [2012a]. *DCMI Metadata Terms*. Dublin Core Metadata Initiative Usage Board. June 14, 2012. http://dublincore.org/documents/2012/06/14/dcmi-terms (cited on page 44).

DCMI [2012b]. *DCMI Metadata Terms: Title Property in Elements Namespace*. Dublin Core Metadata Initiative Usage Board. June 14, 2012. http://dublincore.org/documents/2012/06/14/dcmi-terms/#elements-title (cited on page 49).

DCMI [2012c]. *DCMI Metadata Terms: Title Property in Terms Namespace*. Dublin Core Metadata Initiative Usage Board. June 14, 2012. http://dublincore.org/documents/2012/06/14/dcmi-terms/#terms-title (cited on page 49).

DCMI [2013]. *FAQ/DC and DCTERMS Namespaces*. Dublin Core Metadata Initiative Wiki. Feb. 24, 2013. http://wiki.dublincore.org/index.php/FAQ/DC_and_DCTERMS_Namespaces (cited on page 49).

Eades, Peter [1992]. "Drawing Free Trees". *Bulletin of the Institute of Combinatorics and its Applications* 5 (May 1992), pages 10–36. ISSN 1183-1278. http://www.informatik.uni-rostock.de/~hs162/treeposter/scans/Eades1992.pdf (cited on pages 7, 11).

Feathers, Michael C. [2004]. *Working Effectively With Legacy Code*. Prentice Hall PTR, Oct. 2, 2004. ISBN 0131177052 (cited on pages 52–54).

Fekete, Jean-Daniel and Catherine Plaisant [2003]. *TreeML DTD*. DTD File. 2003. http://nomencurator.org/InfoVis2003/download/treeml.dtd (cited on pages 36, 49).

Few, Stephen [2013]. *Information Dashboard Design: Displaying Data for At-a-Glance Monitoring*. 2nd edition. Analytics Press, 2013. ISBN 1938377001 (cited on pages 3, 9, 11, 15).

Fowler, Martin [1999]. *Refactoring: Improving the Design of Existing Software*. Addison Wesley, 1999. ISBN 0201485672 (cited on page 53).

Gamma, Erich et al. [1994]. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994. ISBN 0201633612 (cited on pages 36, 42, 52, 54).

GBA [2014]. *Geologic Units*. Geological Survey of Austria. Aug. 2, 2014. http://resource.geolba.ac.at/GeologicUnit.html (cited on pages 46, 51).

Herman, Ivan, Guy Melançon, and M. Scott Marshall [2000]. "Graph Visualization and Navigation in Information Visualization: A Survey". *IEEE Transactions on Visualization and Computer Graphics* 6.1 (Jan. 2000), pages 24–43. ISSN 1077-2626. doi:10.1109/2945.841119. http://homepages.cwi.nl/~ivan/AboutMe/Publications/StarGraphVisuInInfoVis.pdf (cited on pages 11, 12, 39–41).

Horridge, Matthew and Sean Bechhofer [2011]. "The OWL API: A Java API for OWL Ontologies". *Semantic Web* 2.1 (Nov. 2011), pages 11–21. ISSN 1570-0844. doi:10.3233/SW-2011-0025. http://semantic-web-journal.net/sites/default/files/swj107_2.pdf (cited on page 52).

Hundhammer, Stefan [2006]. *KDirStat*. Sept. 1, 2006. http://kdirstat.sourceforge.net/ (cited on pages 9, 10).

Inkscape [2014]. *Inkscape*. The Inkscape Team. July 18, 2014. http://inkscape.org/ (cited on pages 17, 20, 38).

Isaac, Antoine and Ed Summers, editors [2009]. *SKOS Simple Knowledge Organization System Primer*. W3C Working Group Note. Aug. 18, 2009. http://w3.org/TR/2009/NOTE-skos-primer-20090818 (cited on pages 43, 44, 52).

James, Mark [2006]. *Silk Icons*. Mar. 12, 2006. http://famfamfam.com/lab/icons/silk (cited on page 57).

JAXB [2014]. *JAXB*. JAXB Expert Group. July 18, 2014. https://jaxb.java.net/ (cited on page 54).

Jupp, Simon [2013]. *SKOS API*. June 8, 2013. http://github.com/simonjupp/java-skos-api (cited on page 50).

Jupp, Simon, Sean Bechhofer, and Robert Stevens [2009]. "A Flexible API and Editor for SKOS". In: *Proc. 6th European Semantic Web Conference on The Semantic Web: Research and Applications (ESWC 2009)*. (Heraklion, Crete, Greece). Berlin, Heidelberg: Springer, 2009, pages 506–520. ISBN 3642021204. doi:10.1007/978-3-642-02121-3_38 (cited on page 50).

KDEWiki [2011]. *Filelight*. KDE UserBase Wiki. June 24, 2011. http://userbase.kde.org/Filelight (cited on pages 11, 12).

Kruja, Eriola et al. [2002]. "A Short Note on the History of Graph Drawing". In: *Proc. International Symposium on Graph Drawing (GD 2002)*. (Vienna, Austria). Springer LNCS 2265. 2002, pages 272–286. doi:10.1007/3-540-45848-4_22. http://merl.com/publications/docs/TR2001-49.pdf (cited on page 6).

Lima, Manuel [2014]. *The Book of Trees: Visualizing Branches of Knowledge*. Princeton Architectural Press, May 1, 2014. ISBN 1616892188 (cited on pages 6, 7).

Megginson, David [2004]. *SAX*. Apr. 27, 2004. http://www.saxproject.org/ (cited on page 53).

Meirelles, Isabel [2013]. *Design for Information*. Rockport, 2013. ISBN 1592538061 (cited on pages 6, 15).

Miles, Alistair and Sean Bechhofer, editors [2009]. *SKOS Simple Knowledge Organization System Reference*. W3C Recommendation. Aug. 18, 2009. http://w3.org/TR/2009/REC-skos-reference-20090818 (cited on pages 1, 36, 43).

Nethercote, Nicholas and Julian Seward [2007]. "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation". In: *Proc. 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2007)*. (San Diego, California, USA). ACM, June 2007, pages 89–100. ISBN 1595936335. doi:10.1145/1250734.1250746. http://valgrind.org/docs/valgrind2007.pdf (cited on page 13).

Nguyen, Trong Dung, Tu Bao Ho, and Hiroshi Shimodaira [2000]. "A Visualization Tool for Interactive Learning of Large Decision Trees". In: *Proc. 12th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2000)*. IEEE, Nov. 2000, pages 28–35. doi:10.1109/TAI.2000.889842. http://dspace.jaist.ac.jp/dspace/bitstream/10119/4655/1/9819.pdf (cited on pages 7, 11).

Nussbaumer, Alexander [2005]. "Hierarchy Browsers: Integrating Four Graph-Based Hierarchy Browsers into the Hierarchical Visualisation System (HVS)". Master's thesis. Graz University of Technology, Austria, Aug. 23, 2005. http://ftp.iicm.tugraz.at/pub/theses/anuss.pdf (cited on pages 34–37, 61–63).

Oracle [2014]. *Java*. Sept. 10, 2014. http://oracle.com/technetwork/java (cited on page 35).

Otlet, Paul [1934]. *Traité de documentation: le livre sur le livre, théorie et pratique*. Editions Mundaneum, 1934. https://archive.org/details/OtletTraitDocumentationUgent (cited on pages xi, 15, 16).

OWL [2014]. *OWL API*. University of Manchester. June 23, 2014. http://owlapi.sourceforge.net/ (cited on page 52).

Penner, Robert [2002]. *Robert Penner's Programming Macromedia Flash MX*. McGraw Hill, Oct. 24, 2002. ISBN 0072223561 (cited on page 41).

Penz, Peter [2012]. *Dolphin File Manager*. June 26, 2012. http://dolphin.kde.org/ (cited on pages 7, 8).

Pietsch, Theodore W. [2013]. *Trees of Life: A Visual History of Evolution*. Johns Hopkins University Press, May 2, 2013. ISBN 1421411857 (cited on page 6).

Putz, Werner [2005]. "The Hierarchical Visualization System: A General Framework for Visualizing Information Hierarchies Using the Example of Information Pyramids". Master's thesis. Graz University of Technology, Austria, Mar. 11, 2005. http://ftp.iicm.tugraz.at/pub/theses/wputz.pdf (cited on pages 35, 42, 49, 50, 61, 64, 65).

Python [2014]. *Python*. Python Software Foundation. Aug. 1, 2014. http://python.org/ (cited on pages 15, 16, 18–20, 22, 24, 25, 28, 30, 31, 34).

Ribon, Aurelien [2012]. *Universal Tween Engine*. July 3, 2012. http://code.google.com/p/java-universal-tween-engine (cited on page 41).

Schulz, Hans-Jörg [2011]. "Treevis.net: A Tree Visualization Reference". *IEEE Computer Graphics and Applications* 31.6 (Oct. 20, 2011), pages 11–15. ISSN 0272-1716. doi:10.1109/MCG.2011.103. http://vcg.informatik.uni-rostock.de/~hs162/pdf/treevisnet.pdf (cited on page 5).

Schulz, Hans-Jörg [2015]. *Treevis.net: A Visual Bibliography of Tree Visualization 2.0*. Mar. 2015. http://treevis.net/ (cited on page 5).

Sedgewick, Robert and Kevin Wayne [2011]. *Algorithms*. 4th edition. Addison-Wesley, Mar. 9, 2011. ISBN 032157351X (cited on page 49).

Sheth, Nihar and Qin Cai [2003]. *Visualizing MeSH Dataset using Radial Tree Layout*. Apr. 29, 2003. http://iv.slis.indiana.edu/sw/papers/radialtree.pdf (cited on pages 25, 26).

Skiena, Steven S. [2008]. *The Algorithm Design Manual*. 2nd edition. Springer, 2008. ISBN 1848000693 (cited on pages 7, 11).

Spence, Robert [2007]. *Information Visualization: Design for Interaction*. 2nd edition. Pearson, 2007. ISBN 0132065509 (cited on page 3).

Stasko, John and Eugene Zhang [2000]. "Focus+Context Display and Navigation Techniques for Enhancing Radial, Space-Filling Hierarchy Visualizations". In: *Proc. 6th IEEE Symposium on Information Visualization (InfoVis 2000)*. (Salt Lake City, Utah). IEEE, Oct. 2000, pages 57–65. doi:10.1109/INFVIS.2000.885091. http://cc.gatech.edu/~john.stasko/papers/infovis00.pdf (cited on page 11).

Tantau, Till and Christian Feuersaenger [2014]. *PGF and TikZ – Graphic systems for TeX*. Mar. 30, 2014. http://sourceforge.net/projects/pgf (cited on pages 7, 8).

TreeML [2006]. *TreeML Specification*. Marlboro College Wikiacademia. Dec. 18, 2006. http://cs.marlboro.edu/courses/fall2006/tutorials/information_visualization/TreeML (cited on pages 36, 49).

Valgrind [2013]. *Valgrind*. Valgrind Developers. 2013. http://valgrind.org/ (cited on page 13).

Van Wijk, Jarke J. and Wim A. A. Nuij [2003]. "Smooth and Efficient Zooming and Panning". In: *Proc. 9th IEEE Symposium on Information Visualization (InfoVis 2003)*. (Seattle, Washington). IEEE, Oct. 2003, pages 15–22. ISBN 0780381548. doi:10.1109/INFVIS.2003.1249004. http://www.win.tue.nl/~vanwijk/zoompan.pdf (cited on page 41).

Van Wijk, Jarke J. and Wim A. A. Nuij [2004]. "A Model for Smooth Viewing and Navigation of Large 2D Information Spaces". *IEEE Transactions on Visualization and Computer Graphics* 10.4 (July 2004). ISSN 1077-2626. doi:10.1109/TVCG.2004.1. http://www.win.tue.nl/~vanwijk/zptvcg.pdf (cited on page 41).

Walker II, John Q. [1990]. "A Node-Positioning Algorithm for General Trees". *Software Practice and Experience* 20.7 (July 1990), pages 685–705. ISSN 0038-0644. doi:10.1002/spe.4380200705 (cited on pages 7, 8, 34, 37, 56).

Ward, Matthew, Georges Grinstein, and Daniel Keim [2010]. *Interactive Data Visualization: Foundations, Techniques, and Applications*. A K Peters, 2010. ISBN 1568814739 (cited on pages 3, 5, 9).

Weidendorfer, Josef [2013]. *KCachegrind*. Apr. 5, 2013. http://kcachegrind.sourceforge.net/ (cited on page 13).

Weidendorfer, Josef, Markus Kowarschik, and Carsten Trinitis [2004]. "A Tool Suite for Simulation Based Analysis of Memory Access Behavior". In: *Proc. 4th International Conference on Computational Science (ICCS 2004)*. (Krakow, Poland). Springer, June 2004, pages 440–447. ISBN 354022114X. doi:10.1007/978-3-540-24688-6_58. http://valgrind.org/docs/callgrind2004.pdf (cited on page 13).

Wheeler, David A. [2004]. *SLOCCount*. 2004. http://dwheeler.com/sloccount (cited on page 54).

Yee, Ka-Ping et al. [2001]. "Animated Exploration of Dynamic Graphs with Radial Layout". In: *Proc. 7th IEEE Symposium on Information Visualization (InfoVis 2001)*. (San Diego, California). IEEE, Oct. 2001, pages 43–50. doi:10.1109/INFVIS.2001.963279. http://hci.stanford.edu/courses/cs448b/papers/Yee_AnimRadial_InfoVis01.pdf (cited on pages 7, 42).