Universiteit Utrecht

Master Thesis

Stable Voronoi Treemaps for Software System Visualization

Author:
Rinse van Hees

Supervisors:
dr. Jurriaan Hage
dr. Hans L. Bodlaender

A thesis submitted in fulfillment of the requirements for the degree of Master of Science

in

Software Technology
Department of Information and Computing Sciences

August 2014

UNIVERSITEIT UTRECHT

Abstract

Software Technology
Department of Information and Computing Sciences

Master of Science

Stable Voronoi Treemaps for Software System Visualization

by Rinse van Hees

In this thesis we introduce stable Voronoi treemaps, a visualization technique that is designed for software quality monitoring. When monitoring the quality of a software system the analysis results of multiple versions of a software system have to be interpreted. Stable Voronoi treemaps help with interpreting the analysis results by creating stable and deterministic pictorial representations. This means that insight gained in one pictorial representation can easily be carried over to another.

As part of our implementation of stable Voronoi treemaps we introduce a sweep line algorithm for additively weighted power Voronoi diagrams. The algorithm extends Fortune's algorithm for Voronoi diagrams by adding weights to the Voronoi sites and using the power distance function that takes those weights into account.

To provide stability to Voronoi treemaps we introduce an algorithm based on scaled Hilbert curves that places Voronoi sites in a deterministic manner. By also enforcing a strict order on the data being visualized we can ensure that the pictorial representations remain visually close to each other while clearly showing the difference.

Using an empirical study we validate our result and conclude that stable Voronoi treemaps are useful for software quality monitoring and software quality assessment in general.

Contents

Abstract i

List of Figures v

List of Tables vii

1 Introduction 1
- 1.1 Thesis focus . . . 2
- 1.2 Main contributions . . . 3
- 1.3 Thesis organization . . . 3

2 Background 5
- 2.1 Software visualization . . . 5
  - 2.1.1 The value of visualization . . . 6
  - 2.1.2 Intentions in software visualization . . . 7
    - 2.1.2.1 A condensed version of the visualization "intentions" for experts . . . 7
    - 2.1.2.2 A condensed version of the visualization "intentions" for non-experts . . . 8
  - 2.1.3 Software visualization taxonomies . . . 8
- 2.2 The Software Improvement Group . . . 9
  - 2.2.1 Software Risk Assessment . . . 9
  - 2.2.2 Software Monitoring . . . 10
  - 2.2.3 The SIG visualization niche . . . 11
- 2.3 Software visualization techniques . . . 11
  - 2.3.1 Voronoi treemaps for visualization of software metrics . . . 12
  - 2.3.2 Visualizing test suites to aid in software understanding . . . 12
  - 2.3.3 Java Interactive Visualization Environment . . . 13
  - 2.3.4 SDP layout of high-dimensional data . . . 13
  - 2.3.5 Hierarchical edge bundles . . . 15
  - 2.3.6 A space of layout styles for hierarchical graph models of software systems . . . 16
  - 2.3.7 Visualizing multiple evolution metrics . . . 17
  - 2.3.8 An open visualization toolkit for reverse architecting . . . 17
  - 2.3.9 EVolve: an open extensible software visualization framework . . . 17
  - 2.3.10 Comparison matrix of visualization techniques . . . 20
- 2.4 Chosen visualization . . . 21

2.4.1 Treemaps: an overview . . . 21
2.4.2 Voronoi treemap benefits . . . 23
2.4.3 Voronoi treemap drawbacks . . . 25
2.4.4 Stable Voronoi treemaps . . . 25

3 Sweep line algorithm for additively weighted power Voronoi diagrams 26
3.1 What is a Voronoi diagram? . . . 26
3.1.1 Voronoi diagram definition . . . 27
3.2 Fortune's algorithm . . . 27
3.2.1 Concept behind Fortune's algorithm . . . 28
3.2.2 2D approach . . . 30
3.2.3 Fortune's algorithm . . . 33
3.3 Fortune's algorithm extended to additively weighted power Voronoi diagrams . . . 37
3.3.1 Additively weighted power Voronoi diagram definition . . . 37
3.3.2 Sweep line, beach line and events . . . 39
3.3.2.1 Site events . . . 39
3.3.2.2 Circle events . . . 39
3.3.2.3 Sweep line and beach line . . . 41
3.3.2.4 Top site circle event . . . 43
3.3.2.5 Beach arcs . . . 43
3.4 Algorithm limitations . . . 45
3.5 The algorithm . . . 45

4 Voronoi treemaps 50
4.1 Centroidal Voronoi diagrams . . . 50
4.2 Additively weighted power centroidal Voronoi diagrams . . . 51
4.3 Recursively building a Voronoi treemap . . . 53

5 Stable Voronoi treemap algorithm 58
5.1 Voronoi site ordering . . . 58
5.2 Hilbert curve Voronoi site placement . . . 59
5.3 Variable Hilbert curve Voronoi site placement . . . 61
5.4 Optimizing stable Voronoi treemap creation . . . 63
5.5 Incremental Voronoi treemap creation . . . 64

6 Empirical validation 70
6.1 Web application . . . 70
6.1.1 Adding analysis results . . . 70
6.1.2 Creating a Voronoi treemap . . . 73
6.1.3 Interacting with a Voronoi treemap . . . 74
6.1.4 Comparing Voronoi treemaps . . . 75
6.2 Validation cases . . . 76
6.2.1 Checkstyle . . . 77
6.2.2 Apache Jackrabbit Core . . . 78
6.3 User sessions . . . 81
6.3.1 Voronoi treemap validation . . . 81
6.3.2 Speed validation . . . 82

6.3.3 Stability validation 82
6.3.4 User validation strengths and limitations 82
6.4 Validation results 83

7 Conclusion and future work **85**
7.1 Sweep line algorithm for additively weighted power Voronoi diagrams . . . 85
7.2 Stable Voronoi treemap algorithm 85

A Implementation details **87**
A.1 Stable Voronoi treemap library 87
A.2 Web application 89

Bibliography **93**

List of Figures

2.1 Cognitive economy model (reproduced from [1]) . . . 6
2.2 Voronoi treemaps (reproduced from [2]) . . . 12
2.3 SDR framework (reproduced from [3]) . . . 13
2.4 JIVE, an object diagram (reproduced from [4]) . . . 14
2.5 JIVE, a sequence diagram (reproduced from [4]) . . . 14
2.6 SDP layout of high-dimensional data (reproduced from [5]) . . . 15
2.7 Zoomed in view of SDP layout of high-dimensional data (reproduced from [5]) . . . 15
2.8 Hierarchical edge bundles, a circular view (left) and a treemap view (right) (reproduced from [6]) . . . 16
2.9 Hierarchical graph model using the degree of clustering layout (reproduced from [7]) . . . 18
2.10 Kiviat diagram visualizing multiple evolution metrics (reproduced from [8]) 19
2.11 Open visualization toolkit (reproduced from [9]) . . . 19
2.12 EVolve (reproduced from [10]) . . . 20
2.13 Tree diagram and corresponding treemap (reproduced from [11]) . . . 22
2.14 Original treemap implementation (reproduced from www.cs.umd.edu) . . . 22
2.15 Squarified treemaps (reproduced from [11]) . . . 23
2.16 Cushioned treemaps (reproduced from [11]) . . . 24
2.17 A Voronoi treemap (reproduced from [2]) . . . 24
2.18 Comparison of two Voronoi treemaps of the same system (reproduced from [12]) . . . 25

3.1 Voronoi diagram . . . 28
3.2 Cones and sweep plane (reproduced from [13]) . . . 29
3.3 Fortune's sweep line algorithm . . . 31
3.4 Beach line is equidistant from the sweep line and the Voronoi sites . . . 32
3.5 Site event: a new arc is added to the beach line (adapted from [14]) . . . 32
3.6 Circle event: an arc is removed from the beach line (adapted from [14]) . . . 33
3.7 Power distance from point P to site O with weight $= r^2$ (reproduced from wikipedia.org) . . . 38
3.8 For any point p on the radical axis, tangents drawn from it to C_1 and C_2 are of equal length, i.e. $PA = PB$ (reproduced from cuemath.com) . . . 38
3.9 Voronoi diagram (left) and AWP Voronoi diagram (right) of the same set of sites . . . 39
3.10 An AWP Voronoi diagram of three sites, one site with a large weight and its corresponding circle and two sites that lie lower than the top of that circle . . . 40

3.11 The radical center and the orthogonal circle of three circles corresponding to weighted Voronoi sites (reproduced from wikipedia.org) . . . 40
3.12 Beach line and sweep line with a site with a large weight that is yet to be encountered . . . 41
3.13 Updated beach line with bisectors . . . 42
3.14 Updated sweep line with a site that is yet to be encountered . . . 43
3.15 The parabolas of two Voronoi sites when the sweep line is below the site, but not below the corresponding circle . . . 43

4.1 Several iterations of Lloyd's algorithm . . . 52
4.2 Moving site s without changing its original weight would mean that site v lies within the circle belonging to s. (reproduced from [12]) . . . 54
4.3 If we increase the weight of site v and only take into account its neighbors in the AWP Voronoi diagram, then the circle belonging to v might overlap with site s. This is prevented by using the neighbors in the standard Voronoi diagram (dotted blue line). (reproduced from [12]) . . . 54

5.1 Hilbert curves of order 1, 2, and 3 (reproduced from opengl.org) . . . 59
5.2 A Hilbert curve intersected with a polygon . . . 60
5.3 A Hilbert curve intersected with a polygon and four Voronoi sites placed at equal intervals . . . 60
5.4 A heat map of the Euclidean distance between any two points on a Hilbert curve of order 8 (reproduced from datagenetics.com) . . . 62
5.5 Three superimposed Hilbert curves of order 1, 2 and 3 (reproduced from wolfram.com) . . . 63
5.6 Three new points (blue) added to the Hilbert curve of order 1 (adapted from opengl.org) . . . 64
5.7 Inserting a new Voronoi site on the edge of the regions of two Voronoi sites 67

6.1 Web application project overview . . . 70
6.2 Web application release overview . . . 71
6.3 Web application adding analysis results . . . 73
6.4 Web application analysis results overview . . . 74
6.5 Available size attributes in web application "Add visualization" . . . 74
6.6 Available sort attributes in web application "Add visualization" . . . 74
6.7 Voronoi treemap at several zoom levels . . . 75
6.8 A tooltip shown when hovering over a Voronoi treemap . . . 75
6.9 The difference between Apache Jackrabbit 1.5.0 and 1.6.0 . . . 76
6.10 The difference between Apache Jackrabbit 2.0.0-beta1 and 2.0.0-beta3 . . . 77
6.11 Checkstyle 5.4 . . . 78
6.12 Apache Jackrabbit 2.0.0-beta1 . . . 79
6.13 Apache Jackrabbit 2.0.0-beta3 . . . 79
6.14 Apache Jackrabbit 2.0.0-beta5 . . . 80
6.15 Apache Jackrabbit 2.0.0 . . . 80

A.1 Stable Voronoi treemap module . . . 88
A.1 Stable Voronoi treemap module continued . . . 89
A.2 Web application back-end . . . 91
A.3 Web application front-end . . . 92

List of Tables

2.1 Comparison matrix of visualization techniques 21

6.1 Overview of user sessions 81

A.1 Libraries and frameworks used in the web application 90

List of Algorithms

1 Fortune's sweep line algorithm for Voronoi diagrams 36

- Procedure HANDLESITEEVENT(s_i) 36

- Procedure HANDLECIRCLEEVENT(s_i) 37

2 Sweep line algorithm for AWP Voronoi diagrams 45

- Procedure HANDLESITEEVENT(se, B, CQ, D, L) 46

- Procedure HANDLECIRCLEEVENT(ce, B, CQ, D) 47

- Procedure HANDLETSCEVENT(te, B, L) 48

- Procedure FINDNEXTCIRCLEEVENT(CQ, L) 48

- Procedure SETARCSITEREFERENCE(te, a, L) 49

3 Lloyd's algorithm for computing centroidal Voronoi diagrams 51

4 Compute Voronoi treemap (single layer) 55

- Procedure ADAPTPOSITIONSANDWEIGHTS(D, S) 55

- Procedure ADAPTWEIGHTS(D, S, P) 56

5 Voronoi treemap creation 57

6 Hilbert curve Voronoi site creation 61

7 Scaled Hilbert curve Voronoi sites 62

8 Variable Hilbert curve Voronoi sites 65

- Procedure FINDPOINTFORNODE(P, r, n, $fraction$, l) 66

- Procedure FINDPOINTFORNODEINCURVE(HC, P, r, n, $fraction$, i_{max}) 67

9 Compute Voronoi treemap (single layer) improved 68

- Procedure COMPUTESTARTWEIGHTS(S, P) 68

- Procedure ADAPTPOSITIONSANDWEIGHTS'(D, S) 69

Chapter 1

Introduction

Software visualization is the process of giving a pictorial representation of a software system. These pictorial representations are used to visualize many aspects of software systems, such as algorithm, control flow and metrics. Every aspect of a software system can be visualized in some way, shape or form, from indentation in integrated development environments to a three-dimensional representation of the control flow.

Software visualizations are often used to give the viewer an easier way of grasping a certain aspect of the software system than looking at the raw source code. When the visualized software system is small, the resulting picture is easy to interpret as a whole. But when software systems get larger, the amount of information present in the visualization grows too. The interpretation gets a lot harder when the viewer is bombarded with sensory data. Several special visualizations have been developed to cope with large amounts of data.

Most current software visualizations are geared towards a one-time visualization of a software system. When visualizing multiple versions of the same software system a new set of problems appears. Consistency of visualizations of different versions of the same software system is needed to guarantee that the viewer is able to take insights from one version and transfer them to another. The changes between versions should be easy to spot in order to allow the viewer to update his insight instead of having to create it anew.

Voronoi treemaps are well suited for visualizing large software systems. Voronoi treemaps use Voronoi tessellation to build very readable and aesthetically pleasing pictorial representations of treelike structures. They are inherently suited for visualizing software

systems, because these systems often have a treelike structure. For example, inheritance and package structures are often treelike. However, the original Voronoi treemap algorithm is not usable when visualizing different versions of the same software system, because it generates wildly different visualizations. We will extend the original Voronoi treemap algorithm in such a way that it can be used to visualize multiple versions of the same software system. We will also extend it so that it can be used in an interactive environment.

1.1 Thesis focus

This work focuses on the development of a software visualization technique that can be used to produce easy to interpret pictorial representations of large software systems. These pictorial representations should not only be easy to interpret, but also be consistent. This means that when two versions of the same software system are visualized, you can easily see that they are related. The visualization technique is used on several different software systems to evaluate the pictorial representations that are produced.

The requirements for the software visualization are as follows.

- The visualization technique should produce pictorial representations of large software systems that allow the viewer to get a better understanding of the software systems.
- The visualization technique should be deterministic: it should produce the same pictorial representation for multiple runs on the same input data.
- The visualization technique should be versatile enough to allow the creation of pictorial representations of different aspects of the same software system.
- Pictorial representations of different aspects of the same software system should be similar enough to identify the same object with ease.
- Pictorial representations of different versions of the same software system should be consistent enough for the viewer to easily identify that it is in fact the same system that is being looked at, unless it has changed significantly.
- Pictorial representations of different versions of the same software system should allow the viewer to identify changes between the versions.

- The visualization technique should be able to create the pictorial representation with reasonable speed.

1.2 Main contributions

In this thesis we introduce stable Voronoi treemaps based on two new algorithms: a sweep line algorithm for additively weighted power Voronoi diagrams and an algorithm to create stable Voronoi treemaps.

Our sweep line algorithm for additively weighted power Voronoi diagrams is based on Fortune's algorithm for Voronoi diagrams. We extend Fortune's algorithm with weighted Voronoi sites and the power distance function.

To create stable Voronoi treemaps we develop an algorithm based on Hilbert curves. We enforce a strict order on the data we visualize and use Hilbert curves to place the data points on the visualization in a deterministic fashion. We also improve the algorithm, based on our observation that we can optimize the placement of data points. This improves the speed of the creation of stable Voronoi treemaps.

1.3 Thesis organization

This thesis builds stable Voronoi treemaps from the bottom up. We start by identifying the benefits and drawbacks of Voronoi treemaps. Then we extend Fortune's algorithm for Voronoi diagrams to be able to create additively weighted power Voronoi diagrams. Next, we build on the extended algorithm to create Voronoi treemaps. Finally, we introduce our algorithms to create stable Voronoi diagrams. We validate our results by using an empirical study.

Chapter 2 introduces the problem domain of the SIG and the value of visualization. We also discuss several visualization techniques and show why Voronoi treemaps are the focus of the rest of this thesis.

Chapter 3 builds a sweep line algorithm for additively weighted power Voronoi diagrams based on Fortune's algorithm. We first show what Voronoi diagrams are and then how they are constructed using Fortune's algorithm. After that, we extend that algorithm for additively weighted power Voronoi diagrams.

Chapter 4 uses the algorithms introduced in Chapter 3 to create Voronoi treemaps. We explain how to recursively build a Voronoi treemap.

Chapter 5 introduces stable Voronoi treemaps. By using deterministic Voronoi site placement we are able to create stable Voronoi treemaps. We also show that based on the new deterministic site placement we can improve the creation time of a Voronoi treemap.

Chapter 6 validates the results of our work in an empirical study. We create a web application that allows users to create and interact with Voronoi treemaps. Using this web application we evaluate the usefulness of stable Voronoi treemaps.

Chapter 7 gives the conclusion of our work. We also discuss possible future work.

Chapter 2

Background

This chapter gives an overview of what software visualization is, how software visualization is used and how the value of the resulting pictorial representations can be determined. The information in this chapter is used as the basis for refining a software visualization technique to suit the needs of the Software Improvement Group (SIG). Section 2.1 explains the concept of software visualization. The Software Improvement Group is introduced in Section 2.2. Section 2.3 gives several examples of software visualization techniques and evaluates them based on criteria from Section 2.1 and Section 2.2.

2.1 Software visualization

Software visualization is a general term that refers to the process of giving a graphical representation of software characteristics and entities. Or as Price et al. put it in [15], "We define [software visualization] as the use of the crafts of typography, graphic design, animation, and cinematography with modern human-computer interaction technology to facilitate both the human understanding and effective use of computer software." This is a very broad definition. It encompasses such things as syntax coloring in modern IDEs and the animation of algorithms. In Subsection 2.1.1, we explore how to determine the value of a visualization. Subsection 2.1.2 identifies several uses of, or intentions in, software visualization. In Subsection 2.1.3, several taxonomies are discussed, which will be used in Subsection 2.2.3 to identify the SIG software visualization niche.

2.1.1 The value of visualization

The value of visualization is expressed in the value of the information that is extracted by the viewer. As Myers et al. said in [16], "Human information processing is clearly optimized for pictorial information, and pictures make data easier to understand for the programmer", we always benefit from creating a pictorial representation of its raw data counterpart. But as Tudoreanu remarked in [17], while humans are good at interpreting pictures, users need to be able to have control over what is displayed. Otherwise, it would be easy to overload the viewer and render the benefits of the visualization void. Van Wijk [1] described the cognitive economy model that depicts the factors that come into play when making good visualizations. In Figure 2.1, we can see that the amount of knowledge gained from the pictorial representation not only depends on the prior knowledge and the perceptual and cognitive ability of the user, but also on the quality of input data and the visualization algorithm.

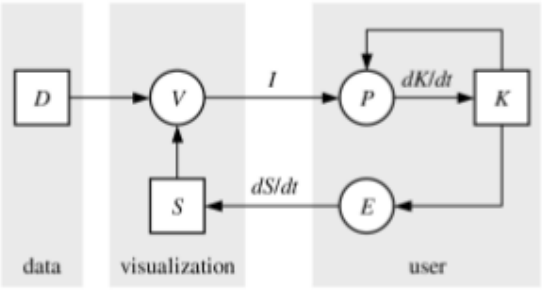

FIGURE 2.1: Cognitive economy model (reproduced from [1])

- V: the actual visualization
- D: the data to be visualized according to specification S (selection of data, algorithm)
- S: the specification of how to transform the data into a picture
- I: the image that is presented to the viewer
- K: the current knowledge of the viewer
- P: the perceptual and cognitive ability of the viewer
- E: controller allowing the viewer to change the view on the data

2.1.2 Intentions in software visualization

The function of software visualization depends on the intentions and needs of the audience. Experts will use visualizations differently from novice programmers or non-experts. Petre, Blackwell and Green [18] identified a set of visualization "intentions" for experts and a set for non-experts. At the end of this subsection, we give a condensed version of the sets of intentions. These sets give a good overview of what purposes visualizations can have. We can use these sets as guidelines when designing and evaluating visualizations. Maletic et al. [19] pointed out that there is no single visualization that can fulfill all purposes; different visualizations are needed to support different purposes. In [20], this is summarized quite nicely:

> Some software visualisation provides an alternative formalism, not a data picture or a machine model, but a regular, symbolic re-presentation of software with a new emphasis, in order to support an otherwise ill-supported style of reasoning. The notion is that an effective display can ease the user's reasoning; the likelihood is that having an effective display changes the user's tactics, if not the nature of the user's reasoning. The display becomes a focus for reasoning, for example by replacing some internal representations with external ones, and hence allowing the user to use different tactics in finding, recalling, examining, or comparing information.

2.1.2.1 A condensed version of the visualization "intentions" for experts

- Externalize images: getting a pictorial representation close to the experts mental model. This can be used in communication with other experts or serve as a backup or external memory so that the expert can come back to it later, without much effort.
- To provide tools for thinking: provide a representation that complements and supplements the experts thinking and giving alternative views or mappings that help the expert to understand the data and navigate through it.
- To harness the computer as a collaborator in problem solving: using the computers computational skills to do part of the reasoning for the expert by programming it. Programming it to detect trends and anomalies, and inform the expert on them.

2.1.2.2 A condensed version of the visualization "intentions" for non-experts

- To provide a glimpse inside the experts mind: to give an impression of the experts reasoning, allow non-experts to understand the experts opinion.
- To teach a non-expert: to show how an algorithm works by visualizing its stages, to teach a non-expert how to use a programming paradigm and to visualize relations that are not explicit.

2.1.3 Software visualization taxonomies

Software visualization is a large field, and to bring some order to the field several taxonomies have been presented. One of the first taxonomies was presented by Myers [21] and has served as basis for other studies. Myers identified three visualization subject categories: code, data and algorithms. Furthermore he identified whether the visualizations are static or dynamic. Stasko [22], Roman [23] and Price et al. [15] extended this basic categorization. Of these extensions the taxonomy of Price et al. is the most extensive and flexible and will be the focus of the rest of this subsection.

Price et al. identified six major categories, each with several minor categories. These minor categories can have subcategories and those subcategories in turn can have subcategories. This makes for a very versatile taxonomy. Each of the major categories can be characterized as follows.

- Scope: What range of programs can the software visualization system take as input for visualization? Does the system allow the user to input his own programs or does it show a set of predefined examples? What restrictions are placed on the visualized program, is concurrency supported, and how large and complex can the program and data set be?
- Content: What subset of information about the software is visualized by the software visualization system? Does the system visualize program structure, code, control flow, data, data flow, or is it an algorithm visualization system? The line between program visualization and algorithm visualization is a fine one. If the system is for teaching an algorithm as opposed to showing the working of an implementation, then it can be said to be algorithm visualization.
- Form: What are the characteristics of the output of the system? To what does the visualization produce its output, is it printed or presented on a screen? Is visualization used to display temporal information? What kind of graphical vocabulary

is used, does the visualization use color, shape or extra dimensions (3D)? Does the system allow you to zoom in or out, show low-level detail or a high-level overview? Can you view the same data from different angles? And can you visualize two distinct programs simultaneously to compare them?

- Method: How is the visualization specified? Does the visualization system require human action to visualize the program, or is it fully automatic? Can the visualizations be scripted and customized? How does the visualization system connect to the program to get the data to visualize?
- Interaction: How does the user of the software visualization system interact with it and how does he/she control it? Is the result of a visualization a static picture or does it allow the user to change the view and navigate through the visualization? Does the system allow the saving and recording of visualizations?
- Effectiveness: How well does the system communicate information to the user? This is a generalization of what has been discussed in Section 2.1.1.

2.2 The Software Improvement Group

The Software Improvement Group (SIG) is an international software consultancy firm. The SIG specializes in analyzing large software systems to assess them on their technical quality and maintainability. The two main services provided by the SIG that give insight in the technical quality of a software system are Software Risk Assessment (SRA) and Software Monitoring Service. During an SRA a software system is subjected to a one-time in-depth analysis of its source code. Based on the analysis results the consultants write a software health report. Software Monitoring also does an analysis of the source code, although it is of a recurring nature. New versions of a software system are analyzed and automated reports are generated based on the analysis results.

2.2.1 Software Risk Assessment

Software systems generally are in use for a long time. During that time the needs and demands of the users of the system will change. The system will have to adapt to these new needs and the system will have to keep evolving as long as it is in use. This means that a software system not only has to do its current job well, it also has to be future-proof. For a software system to be future-proof its most fundamental building blocks must be future-proof. The source code of a software system has to be of high quality;

then it can easily be extended, changed and maintained.

A Software Risk Assessment (SRA) gives insight in the code quality of a software system. During an SRA the source code of a software system is subjected to a detailed and automated analysis. The analysis results are supplemented with interviews and consultations with system experts. All gained insight is compiled into a comprehensive document detailing all the quality aspects of the software system source code.

During an SRA several techniques to visualize the analysis results are used. The resulting visualizations are used by the SIG experts to get an overview of the software system that they are analyzing, but also to report their findings to the customer. Different visualizations are needed for the different audiences. The SIG experts can read call graphs and other technical visualizations. The customers might not always be able or want to do that, because usually they are upper management and have no programming experience. This requires a different set of visualizations: visualizations that give a high-level overview.

Both types of visualizations are not without their problems. When the software system being analyzed is large, the amount of information that needs to be visualized is staggering. Especially the detailed visualizations for the SIG experts suffer from this.

2.2.2 Software Monitoring

Software systems continually evolve to reflect the new needs and demands of the users. Maintainability is a key aspect of the source code of a software system that reflects how future-proof it is. Keeping an eye on software maintainability is a must during development. Software Monitoring allows you to get an ongoing report of your software maintainability as the system evolves.

At set intervals, Software Monitoring analyzes the source code of a software system and gives a detailed report. This, coupled with the older reports, makes it possible to keep track of the software quality as the system is being developed. The reports are available at several levels of detail: upper management gets a bird's-eye view with reported totals; developers and architects can get a more detailed report.

The reports are supplemented with visualizations that allow the viewer to understand the reports more easily. In the case of Software Monitoring it is also important that the visualizations are consistent over the analyses done at intervals.

2.2.3 The SIG visualization niche

The intended audience of the visualizations that the SIG uses can be categorized in two main categories. The reports the SIG writes during an SRA are written to support the upper management of a company to make strategic decisions about the software systems. In these reports a high-level overview is given and little technical detail is presented. On the other hand, you have the SIG consultants that write the reports. They need to get down to the nitty-gritty details of the software system. The software analysis results give them these details, but there are so many details present in the analysis results that finding the details of interest can be quite a chore. Visualization can give the "big picture" in which the SIG consultant can quickly find the areas of interest and zoom in on those.

Software Monitoring has the same two categories of intended audience. The upper management keeps track of the software development using a high-level overview; the programmers and architects use a more low-level view to see the details of what has to be done. The programmers and architects, just like the SIG consultants, can use a high-level overview to drill down to the details that need their attention.

High-level overviews of a software system are relatively easy to generate using existing visualization techniques. Pie charts and graphs can give a reasonable overview of the general state of a software system. But when trying to use those to visualize all the detail that is available in the source code analysis results, they become impossible to read. A visualization that can cope with the amount of analysis results that is generated when analyzing a large software system is a must. In the next section we will show several software visualization techniques that could be used to visualize large amounts of data.

2.3 Software visualization techniques

In this section, we give an overview of several visualization techniques that show potential for the SIG. Each of these visualizations will be introduced and evaluated, using the

criteria that were identified in the previous sections.

2.3.1 Voronoi treemaps for visualization of software metrics

Balzer, Deussen and Lewerentz [2] introduced a variation on traditional treemaps that is more suited for software visualization: Voronoi treemaps. Software systems usually have a rich hierarchical structure; it is not unthinkable that it is twenty levels deep or more. Visualizing metrics in such hierarchies using traditional treemaps has several drawbacks. The subdivision of available space is solely done in one dimension, which results in thin, elongated rectangles with a high aspect ratio. This makes understanding the information much harder. Using polygons instead of rectangles can solve this issue. Using Voronoi tessellation to divide the available space for the top-level and recursing into the other levels will result in a treemap that is easier to read, as can be seen in Figure 2.2. This is due to the fact that the aspect ratio approaches one for each element in the tree, which helps in comparing the relative size of elements.

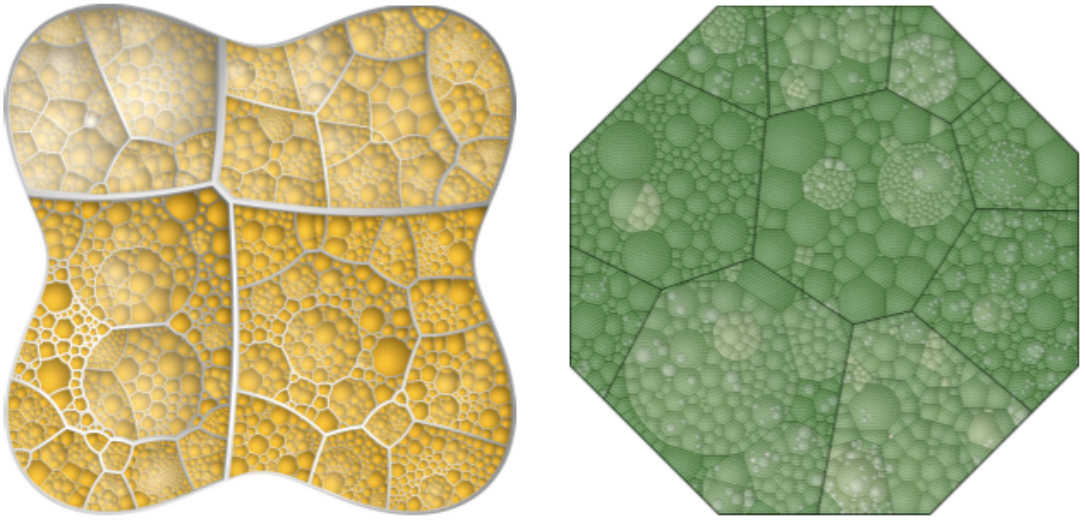

FIGURE 2.2: Voronoi treemaps (reproduced from [2])

2.3.2 Visualizing test suites to aid in software understanding

Test driven software development has gained more support in the last few years. This has as advantage that many newly developed software systems are accompanied by a large set of unit tests. These tests are a good starting point when trying to understand a software system, because they test one single aspect of the total system. Still, you have to browse code to grasp what it does. Visualizing the unit test allows the viewer to easier understand the aspect that is being looked at. Cornelissen et al. [3] developed the SDR framework to do just that: they visualize JUnit test execution traces. Using

a debugger, a profiler, and instrumentation through aspects, they create traces of the JUnit test run. Using filtering they can show different abstraction levels, allowing you to see the nitty-gritty details or a high-level overview of a single aspect within the larger software system. Figure 2.3 shows a visualization of the methods executed during a test run.

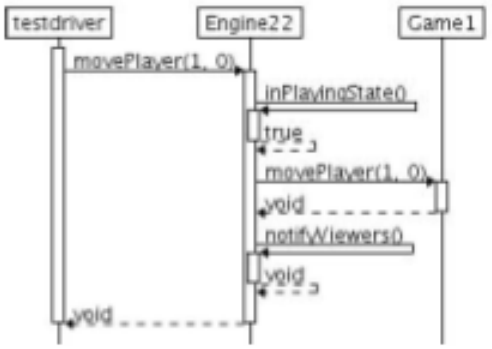

FIGURE 2.3: SDR framework (reproduced from [3])

2.3.3 Java Interactive Visualization Environment

Java Interactive Visualization Environment (JIVE) [4] is a run time visualization and analysis system. It allows for multiple concurrent representations of program state and execution history, has support for forward and reverse execution, and supports graphical queries over program execution. JIVE works on Java programs, using the Java Platform Debugger Architecture. When the state of the subject program changes, the program execution is suspended, the views in JIVE are updated, and the subject program execution is then resumed. As this continues the complete program can be visualized. JIVE has several views on the execution, for example syntax highlighting and highlighting of the statement that is being executed. It also provides a sequence diagram view and a call path view. Figure 2.4 shows an object diagram and Figure 2.5 shows a sequence diagram of the same program execution.

2.3.4 SDP layout of high-dimensional data

In [5], David Gleich et al. describe a method of visualizing relations between music artists based on user preference.

The music rating site Yahoo! Music collects user ratings of artists. If a user gives two or more artists the same rating, then these artists can be said to have a relation. Using the ratings from all users allows Gleich et al. to build a huge set of relations between

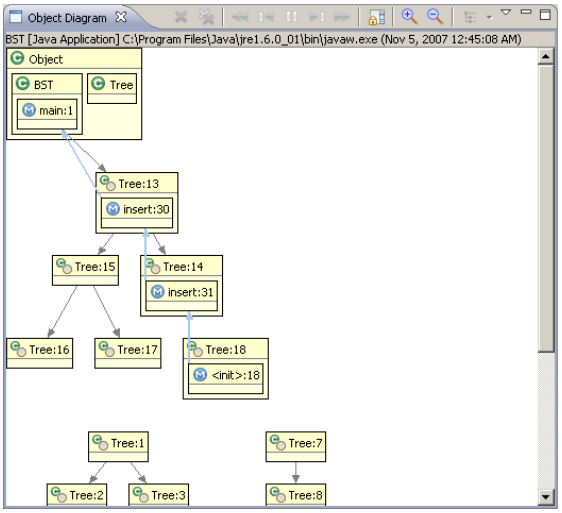

FIGURE 2.4: JIVE, an object diagram (reproduced from [4])

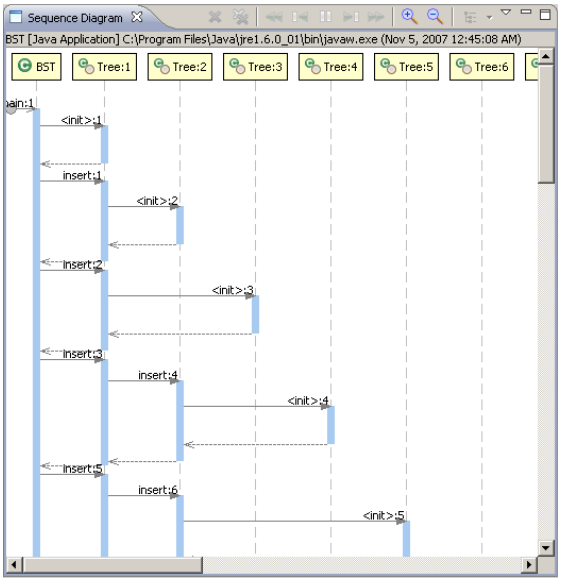

FIGURE 2.5: JIVE, a sequence diagram (reproduced from [4])

artists that can be weighted by the number of users that rate both artists. This can be seen as creating a similarity graph. The similarity graph is used in a non-linear low-dimensional embedding constrained to the surface of a hypersphere. This results in a point cloud, mapped to the surface of the hypersphere, where closely related artists are grouped together. Drawing the twenty strongest relations of each artist gives a view of how groups of artists are connected, without overloading the viewer. Unrolling the spherical representation gives a visually pleasing overview of artist relations, and clearly shows the grouping of styles and which artist bridges the gap between styles. Figure 2.6 shows a high-level overview of the visualization and Figure 2.7 zooms in on the central cluster.

Using the same layout algorithm on call relations between programming entities would

result in a nice overview of a software system that clearly shows which entities are closely related. An alternative approach could be to use the coupling between programming entities as similarity graphs, and visualizing the calls between the entities. This would, in theory, show which entities are closely related but not necessarily call each other.

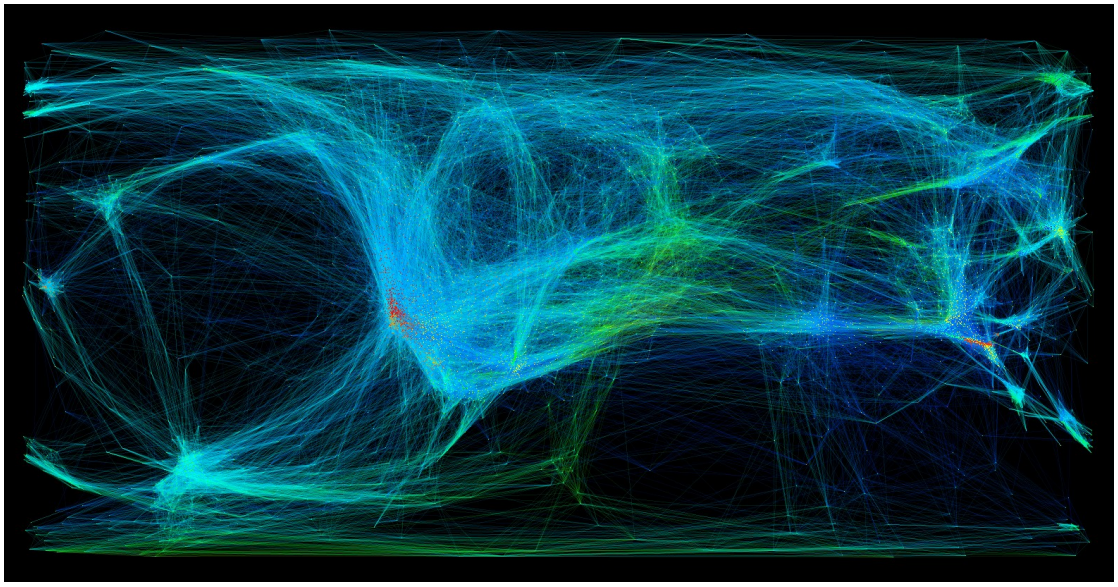

FIGURE 2.6: SDP layout of high-dimensional data (reproduced from [5])

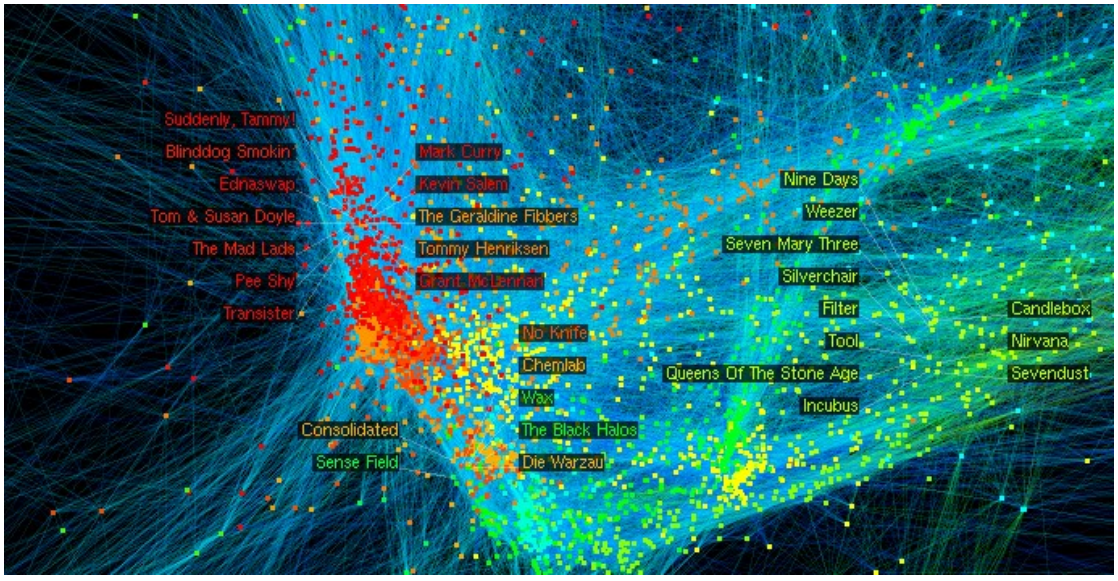

FIGURE 2.7: Zoomed in view of SDP layout of high-dimensional data (reproduced from [5])

2.3.5 Hierarchical edge bundles

Dynamic execution traces are a useful tool when trying to understand the inner working of a complex software system. Such traces tend to get very large; even a simple program

will have thousands of execution steps per trace. Visualizing such traces is a problem; often the visualization will be cluttered and hard to understand. This is partly due to the fact that when many calls from one entity to another occur, the line representing this will obscure other call relations. To solve this, Holten [6] introduced hierarchical edge bundles. When the program entities are placed on the screen, all calls between entities that are visually close to each other will be bundled so that there is less clutter on the screen. In [24] is shown how this can be implemented using a circular bundle view. Entities are placed on the ring of a circle and all calls go through that circle. Figure 2.8 shows an example of a circular bundle view and of hierarchical edge bundles on a treemap.

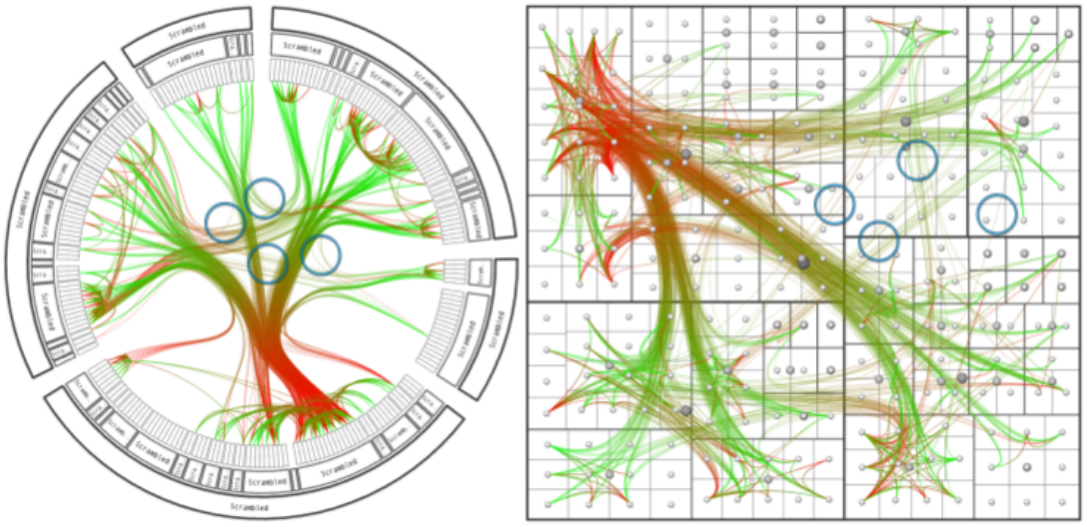

FIGURE 2.8: Hierarchical edge bundles, a circular view (left) and a treemap view (right) (reproduced from [6])

2.3.6 A space of layout styles for hierarchical graph models of software systems

Hierarchical graphs are widely used as models of the structure of software systems. Visualizing these graphs poses some interesting problems though. For example, how do we position the nodes in two- or three-dimensional space? In [7], Noack and Lewerentz tackled this layout problem. For a variety of analyses of the static structure of software systems they derived the requirements for graph layouts that support those analyses. Because no single layout can satisfy all requirements, they introduced a space of layout styles. In this space, the layout styles are organized along the following three dimensions.

- Degree of clustering: layouts with meaningful distances between single nodes (for analyses concerning the local neighborhood of software entities) versus layouts

with meaningful distances between groups of nodes (for analyses concerning the global structure of the software system). Figure 2.9 shows an example.

- Degree of hierarchicalness: layouts that reflect the existence of edges between nodes (for analyses of relationships between software entities) versus layouts that reflect the existence of a common ancestor in the hierarchy tree (for analyses of containment hierarchy).
- Degree of distortion: layouts that faithfully reflect the size of nodes and edges, i.e. the number of low-level nodes and edges that they represent (for analyses with a fairly uniform granularity of the involved entities and relationships) versus layouts where certain edges are magnified (for analyses of details in their global context).

They extended the minimization of energy function, a widely used method for computation of graph layouts, with the degree of clustering, the degree of hierarchicalness, and the degree of distortion. This allows for automatic computation of layouts for analyses.

2.3.7 Visualizing multiple evolution metrics

Software visualization is intrinsically suited for observing the evolution of software systems. Showing the visualization of two versions of the same software system next to each other allows the viewer to determine where things have changed and what has stayed the same. However, there are some challenges, such as that the visualization of both versions should be easily comparable, meaning that objects should be in approximately the same place. Pinzger et al. [8] introduced a system to do just that. The metrics and relations of multiple versions are superimposed on a Kiviat diagram, as seen in 2.10.

2.3.8 An open visualization toolkit for reverse architecting

The open visualization toolkit introduced in [9] is a toolkit geared towards visualizing graphs to facilitate the reconstruction of the architecture of a large software system. The toolkit takes a graph that can have values associated with the nodes as input. The scripting language TCL can be used to make selections and mappings on the graph. The transformed graph is then visualized. There are several predefined visualizations available. Figure 2.11 shows two examples.

2.3.9 EVolve: an open extensible software visualization framework

EVolve [10] is a visualization framework that is not tailored to one specific language, algorithm or analysis, but allows one to plugin new data sources and visualizations.

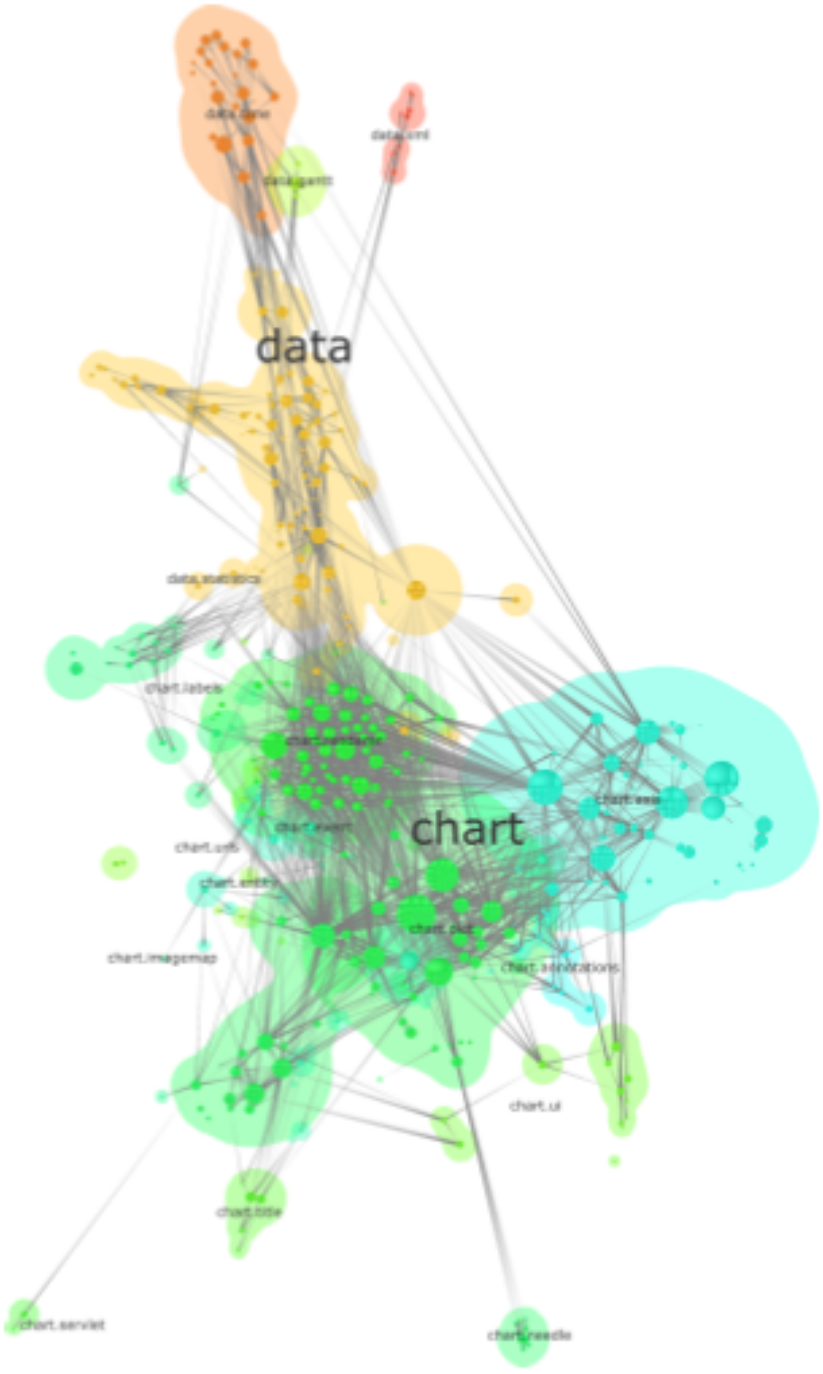

FIGURE 2.9: Hierarchical graph model using the degree of clustering layout (reproduced from [7])

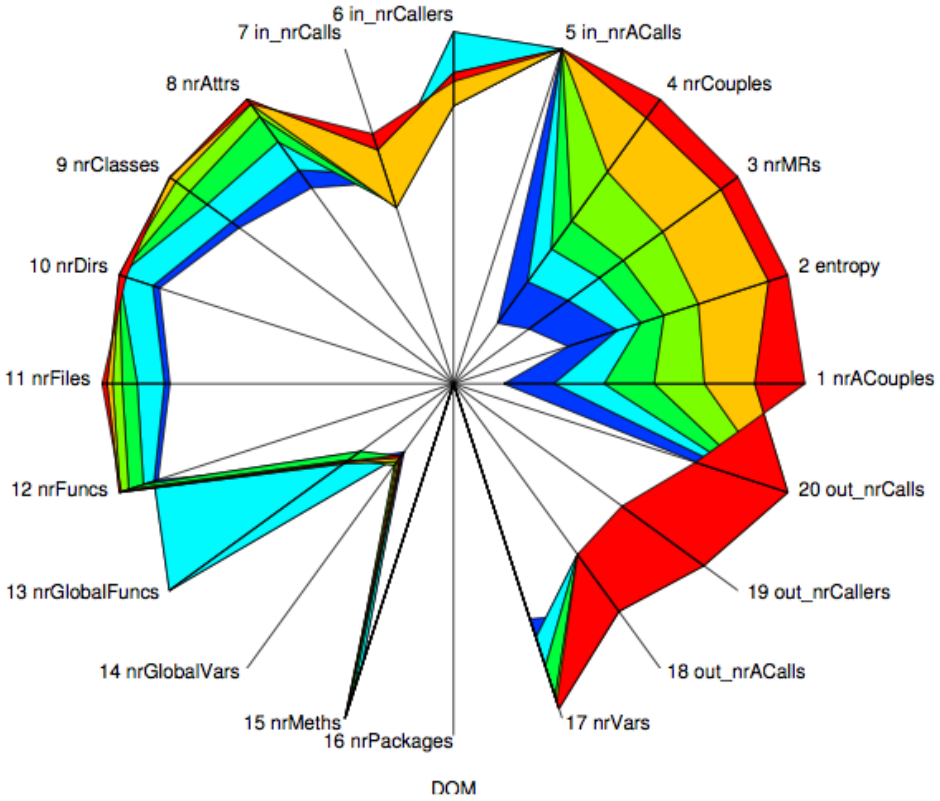

FIGURE 2.10: Kiviat diagram visualizing multiple evolution metrics (reproduced from [8])

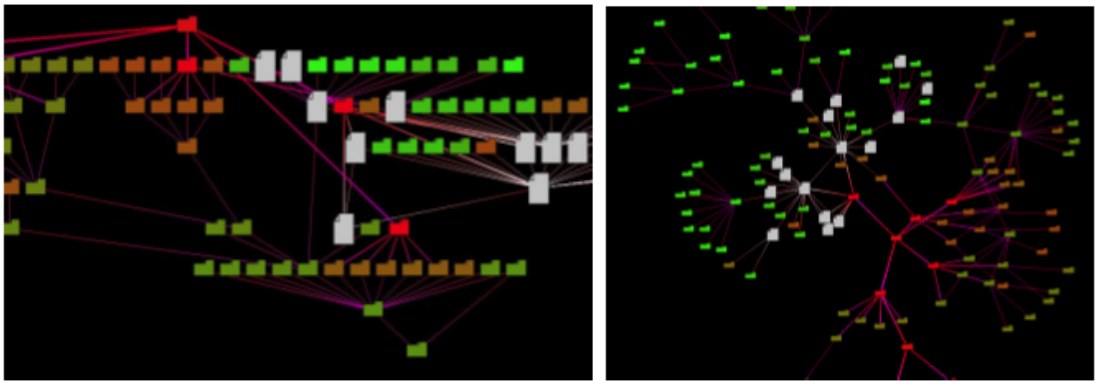

FIGURE 2.11: Open visualization toolkit (reproduced from [9])

This is realized by providing an API to specify visualizations and manipulate data, so that it can be used by the framework. The framework handles all user interaction and the communication between data source and visualizations. Figure 2.12 shows four visualizations of program execution metrics.

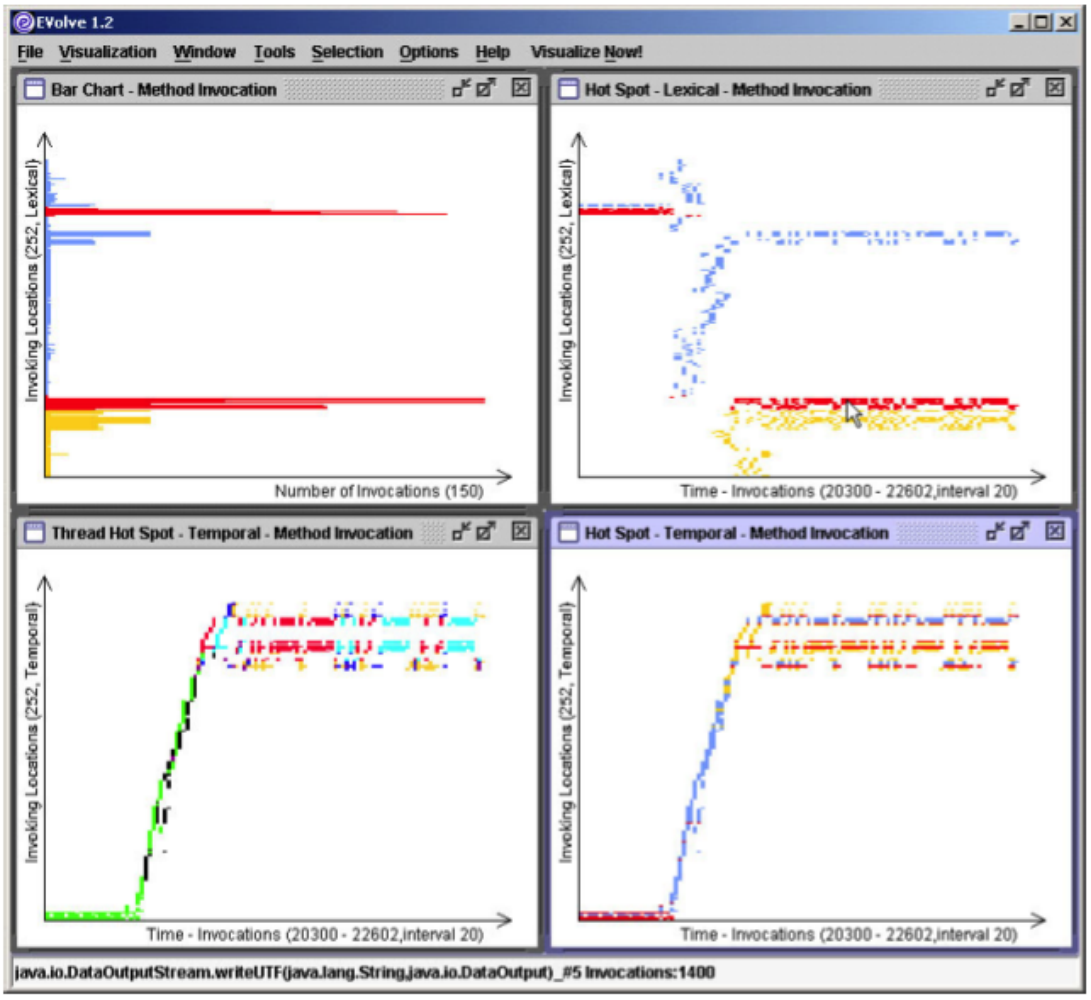

FIGURE 2.12: EVolve (reproduced from [10])

2.3.10 Comparison matrix of visualization techniques

In Table 2.1 we compare the visualization techniques described in this section. We identified several key aspects that a visualization requires to be usable for the intended purposes of the SIG. The visualization techniques are compared using these key aspects. If a visualization has a certain aspect, we mark it with a plus sign (+); if it does not have that aspect, we mark it with a minus-sign (−).

Visualization technique	Section	high-level overview	detailed view	large software systems	interactive	metrics	multiple metrics	relations
Voronoi treemaps	2.3.1	+	+	+	−	+	+	−
SDR framework	2.3.2	−	+	−	−	−	−	+
JIVE	2.3.3	−	+	−	+	−	−	+
SDP layout	2.3.4	+	−	+	−	−	−	+
Hierarchical edge bundles	2.3.5	+	+	+	+	−	−	+
A space of layout styles	2.3.6	+	−	+	−	−	−	+
Multiple evolution metrics	2.3.7	−	+	−	−	+	+	−
Reverse architecting	2.3.8	+	+	+	+	−	−	−
EVolve	2.3.9	−	+	−	+	+	+	−

TABLE 2.1: Comparison matrix of visualization techniques

2.4 Chosen visualization

After looking at the visualizations presented in the previous section and filling in the comparison matrix, we asked the intended users of our tool which visualization would best fit their needs. It was decided that Voronoi treemaps would best fit the needs of the SIG. In this section, we will first give a short history of treemaps and show why they are useful. Then we show the added benefits of Voronoi treemaps. We also discuss the drawbacks of Voronoi treemaps. We then introduce our proposed solution.

2.4.1 Treemaps: an overview

In the early 1990's when 80-megabyte hard disks were still common, Ben Shneiderman became obsessed with creating a compact visualization of directory structures that would show him where his precious hard disk space went. He stumbled upon the idea to split his computer screen into rectangles, in alternating horizontal and vertical directions as you traverse down the directory tree. The size of the rectangles reflected the size of the directories or files on the file system. This solution makes optimal use of the limited screen space available in the 1990's. In 1992, the first paper on the solution by Shneiderman [25] was published. In this paper the term treemap was introduced. It was thought

to best describe the notion of turning a tree into a planer space-filling map, see Figure 2.13. Figure 2.14 shows one of the early treemap visualizations with added color for different file types.

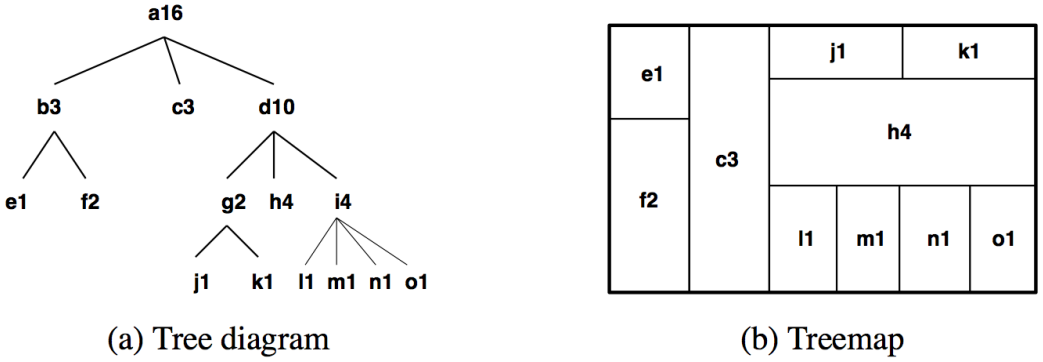

FIGURE 2.13: Tree diagram and corresponding treemap (reproduced from [11])

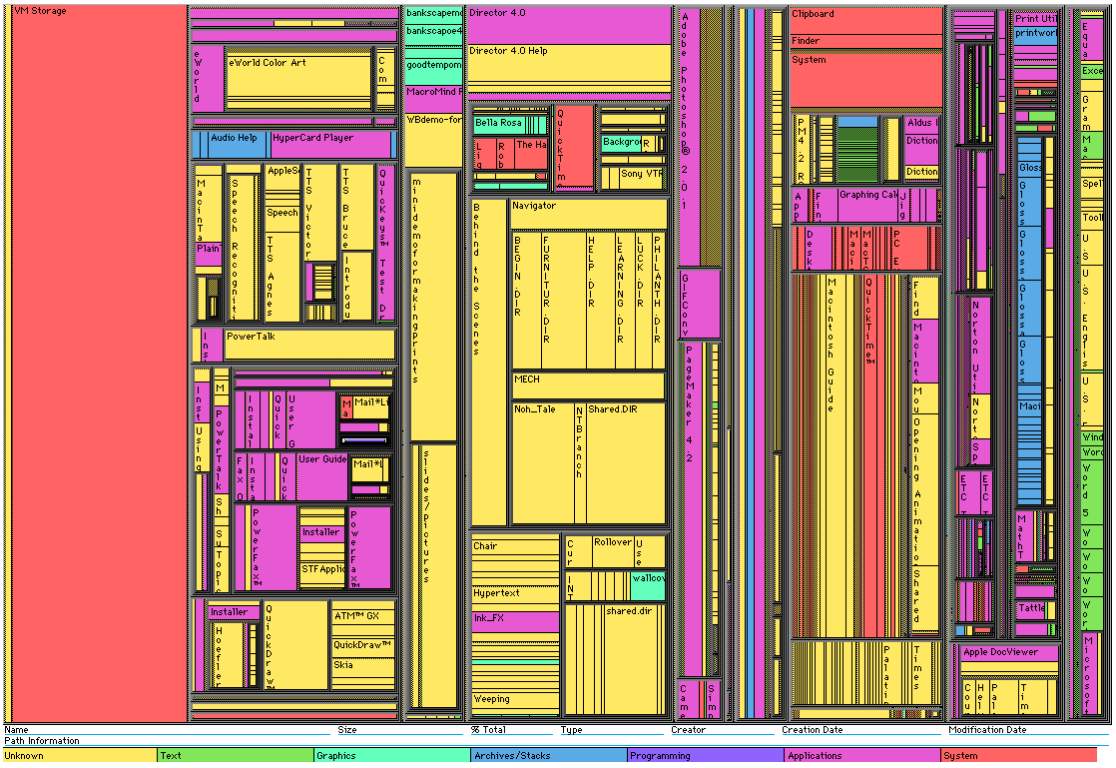

FIGURE 2.14: Original treemap implementation (reproduced from www.cs.umd.edu)

In later years, many extensions of the original treemap algorithm have been made, from adding color to distinguish between file types, to sounds for another characteristic of the file. Treemaps have even been used to visualize stock portfolios. The original algorithm has one shortcoming, one that most extensions inherit: it divides the space for each step solely in one dimension. If many objects or objects with a big difference in size on the same tree level are visualized, this can result in thin elongated rectangles with

a high aspect ratio. Comparing the size of such elongated rectangles and other, not so elongated rectangles, becomes quite a chore. Figure 2.14 shows some good examples of this phenomenon.

Ben Shneiderman and Martin Wattenberg [26] introduced ordered treemaps. Ordered treemaps use a different division algorithm, resulting in rectangles that have an aspect ratio close to one. Squarified treemaps [11] and clustered treemaps [27] each solve the same problem with their own division algorithm. While all these different space division algorithms solve the problem of extreme aspect ratios, they do not tackle the problem of hierarchy visualization. For squarified treemaps we can see this in Figure 2.15. In the original treemap implementation hierarchy was reasonably visible, because of alternating the horizontal and vertical division for each level. In the new algorithms this is lost and consequently the hierarchical element of the visualized objects is lost or at least hard to discern. Van Wijk and van de Wetering [28] solved this by introducing cushioned treemaps. Cushioned treemaps add shading to each rectangle in such a way that it is easier to distinguish the parent-child relationships in the tree. Figure 2.16 shows the same tree as Figure 2.15, but using the cushioned treemap method.

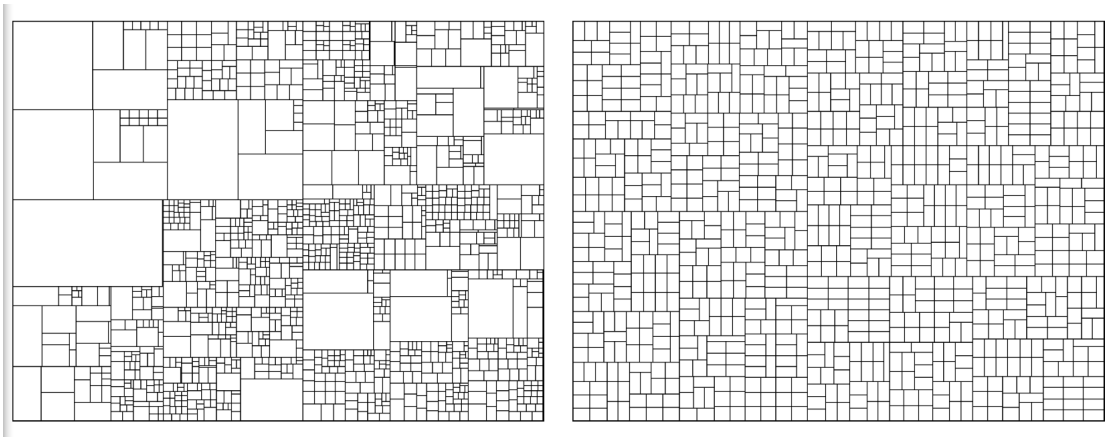

FIGURE 2.15: Squarified treemaps (reproduced from [11])

2.4.2 Voronoi treemap benefits

Voronoi treemaps [2] are another way of solving both the space division problem and the hierarchy visibility problem. Instead of using rectangles, Voronoi treemaps use polygons to divide the screen space. Voronoi treemaps allow for easy distinction of level, because the edges that bound an object in the tree do not line up. This means that an edge of a parent object will not line up with one of its children, or even with the edges of any of its neighbors. Because of the way the polygons are constructed, the aspect ratio

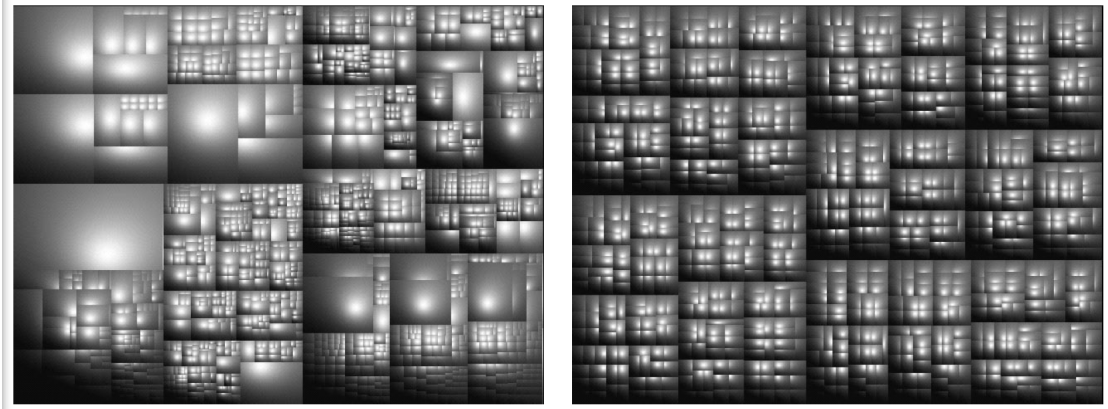

FIGURE 2.16: Cushioned treemaps (reproduced from [11])

approaches one. This makes it easy to compare the size of objects. Figure 2.17 shows an example of a Voronoi treemap.

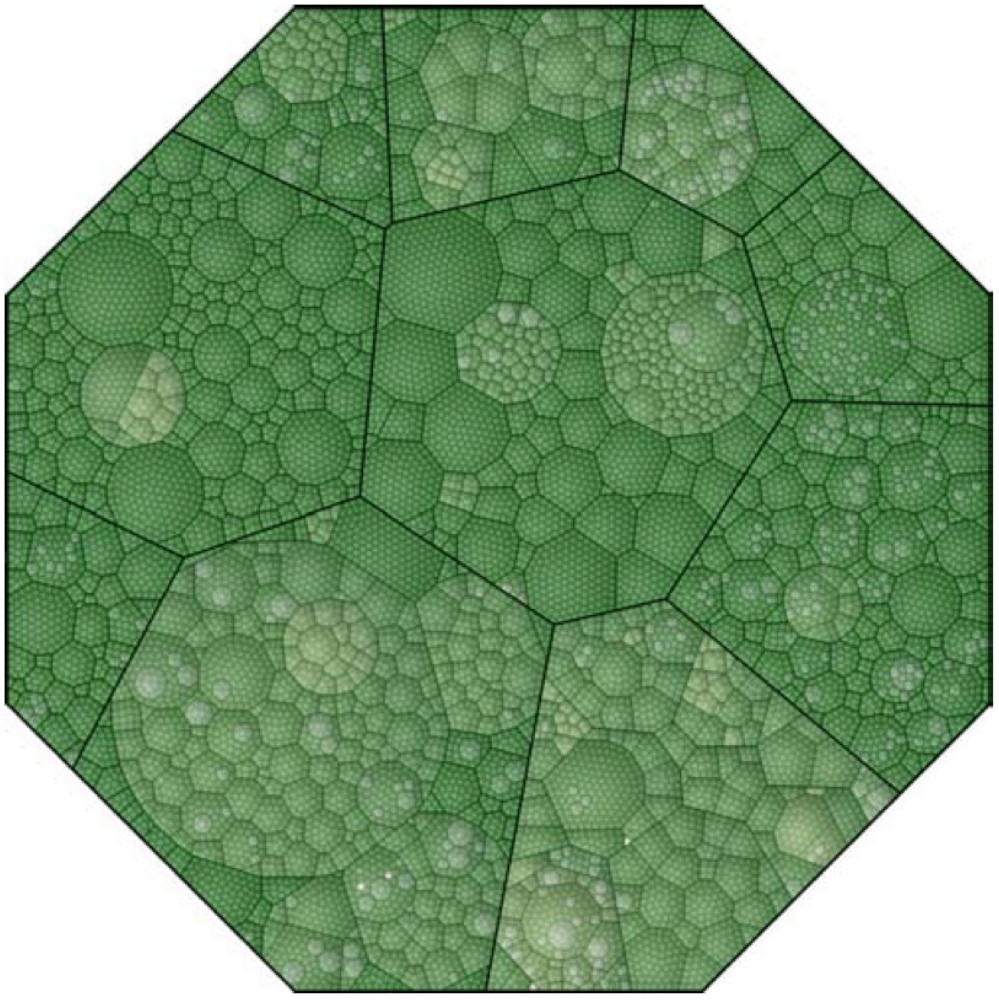

FIGURE 2.17: A Voronoi treemap (reproduced from [2])

2.4.3 Voronoi treemap drawbacks

Voronoi treemaps as they are presented in [2] have one major drawback when visualizing the same data multiple times. Because the algorithm is based on a random initialization, each visualization is very different. This means that in its current form Voronoi treemaps can not be used to visualize the same data set multiple times. We also can not use these Voronoi treemaps to visualize different versions of the same system, see Figure 2.18.

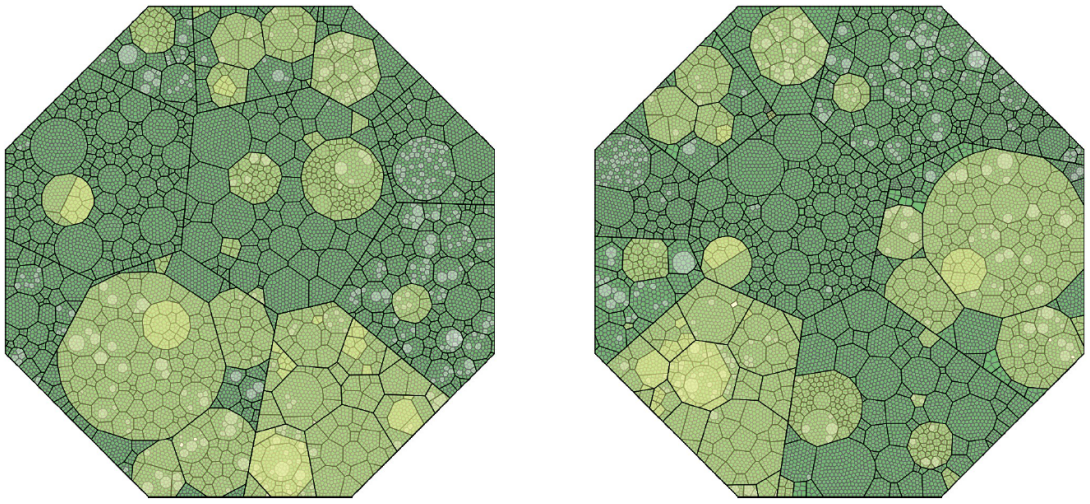

FIGURE 2.18: Comparison of two Voronoi treemaps of the same system (reproduced from [12])

2.4.4 Stable Voronoi treemaps

To remove the drawbacks currently present in Voronoi treemaps, we introduce stable Voronoi treemaps. Stable refers to the fact that the visualization of a system will always be the same when the same data is visualized. We also extend this stabilized version of Voronoi treemaps to handle visualizing multiple versions of the same system. Creating the algorithms for stable Voronoi treemaps is the focus of the rest of this thesis.

Chapter 3

Sweep line algorithm for additively weighted power Voronoi diagrams

Treemaps are a space-filling recursive subdivision of a given area without gaps or overlap. In Voronoi treemaps this recursive subdivision is based on Voronoi diagrams. Voronoi diagrams divide an area into Voronoi regions based on a set of points, called sites. Each Voronoi region corresponds to a site and consists of all points closer to the site than to any other site.

In treemaps the size of a subdivision represents some metric. Voronoi diagrams traditionally are not created with a predefined size for a Voronoi region. This means that to use Voronoi diagrams as basis for treemaps, there needs to be a way to influence the size of the Voronoi regions. Additively weighted power Voronoi diagrams allow for a greater degree of influence on the area of a Voronoi region. The Voronoi site has a weight in addition to a location.

In this chapter, we first introduce Voronoi diagrams and show how they are constructed using Fortune's algorithm. Then we introduce our sweep line algorithm for additively weighted power Voronoi diagrams based on Fortune's algorithm.

3.1 What is a Voronoi diagram?

Voronoi diagrams are named after their inventor, the Russian mathematician Georgi Voronoi [29]. Voronoi diagrams have many uses, from biology to computer science. For example, Voronoi diagrams are used to model crystal growth and to visualize cell growth.

In computer science, Voronoi diagrams are used to compute shortest path and traveling salesman solutions.

3.1.1 Voronoi diagram definition

A Voronoi diagram of a set of points, called sites, divides a plane into regions. Each region corresponds to one of the sites and consists of all points closer to its site than to any other site. In this thesis we will use the following definitions.

- Site: a point in the Euclidean plane that defines a Voronoi region.
- Voronoi region of site s: the set of points closer to s than to any other site.
- Edge: the ray or line segment that bisects two neighboring sites in the Voronoi diagram.
- Bisector: the perpendicular bisector of line segment $\overline{pq}$, where p and q are points.
- Vertex: a point that is equidistant from three or more sites.
- Voronoi tessellation: the process of dividing the plane into Voronoi regions.
- Voronoi diagram: the set of all Voronoi regions and their edges.

The generic definition of a Voronoi diagram of a set S is as follows. Let $S = \{s_1, s_2, \ldots s_n\}$ be a set of n distinct sites in $\mathbb{R}^2$. The Voronoi diagram of S is the partitioning of the plane into n Voronoi regions and their edges, see Figure 3.1. The region of site s_i consists of all points $q \in \mathbb{R}^2$ where $dist(q, s_i) < dist(q, s_j)$ for each $s_j \in S$ with $j \neq i$. The function $dist(p, q)$, where $p = (p_x, p_y)$ and $q = (q_x, q_y)$, is the Euclidean distance function $dist(p, q) = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$. Two adjacent regions belonging to sites s_i and s_j share an edge that is equidistant from s_i and s_j. This edge is a segment of the perpendicular bisector of the line segment $\overline{s_i s_j}$. The edge can be bounded at both ends, bounded at one end or unbounded. A point on the plane that is equidistant from three or more sites is called a vertex. A vertex is the begin or end point of an edge that bisects two sites that are equidistant from the vertex.

3.2 Fortune's algorithm

Fortune's algorithm was introduced in 1987 in his paper *A sweep line algorithm for Voronoi diagrams* [30]. In Subsection 3.2.1, we will introduce the concept behind this

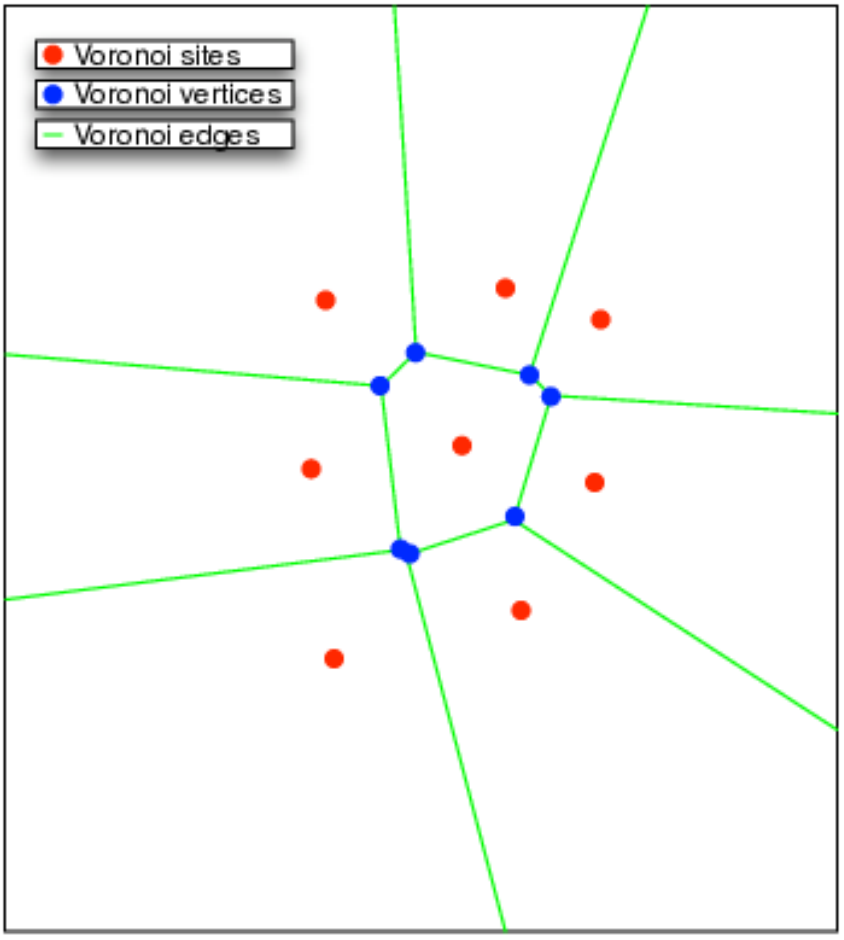

FIGURE 3.1: Voronoi diagram

algorithm by introducing a three-dimensional version. In Subsection 3.2.2, we abstract away from the three-dimensional algorithm and introduce a two-dimensional approach. Then we go one step further in Subsection 3.2.3 and introduce the actual algorithm.

3.2.1 Concept behind Fortune's algorithm

Fortune's algorithm is a sweep line algorithm. Sweep line algorithms use a conceptual horizontal line to sweep over the plane from top to bottom. As the sweep line moves over the plane, we keep track of what it encounters to be able to compute the desired structure. To be more concise, information is maintained about the intersection of the structure and the sweep line. In traditional sweep line algorithms, everything above the sweep line is fully computed and independent from everything below the sweep line.

To introduce the sweep line algorithm for Voronoi diagrams, we will use a three-dimensional approach instead of the two-dimensional Euclidean plane. We place cones with a 45-degree angle above all sites, see Figure 3.2. Because of the shape and placement of each cone we can now easily determine the bisector of two sites by looking where the corresponding cones intersect. The projection of the intersection of two cones on the ground plane corresponds to the bisector of the two sites that the cones belong to. By using this fact, we can construct a sweep line algorithm. Because we are working in a three-dimensional space, we will use a plane instead of a line. For reasons that will become obvious later in this subsection, the sweep plane has a 45-degree angle to the ground plane, as seen in Figure 3.2.

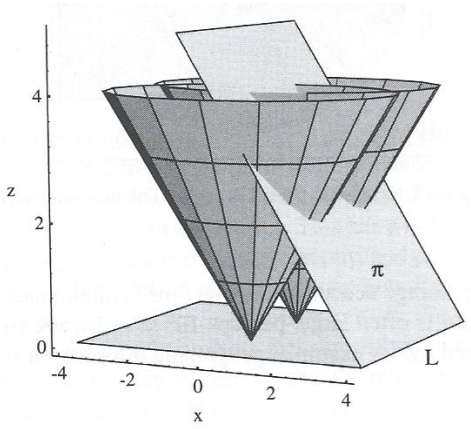

FIGURE 3.2: Cones and sweep plane (reproduced from [13])

When we sweep the plane over the ground plane, we keep track of all places the sweep plane intersects with two or more intersecting cones. This way we have all bisectors between sites that the sweep plane passed. However, we are not interested in all bisectors, but only in the segments of bisectors that are actually edges in the Voronoi diagram. The bisector segments that are edges in the Voronoi diagram can be found by only considering the cone intersections that do not lie inside another cone.

By projecting the relevant intersections encountered by the sweep plane onto the ground plane we create the Voronoi diagram. This shows that by using the three-dimensional sweep plane approach we can construct a Voronoi diagram. However, the use of cones and a sweep plane introduces a lot of complex math that we can abstract away from. In the next subsection, we will do just that and introduce a two-dimensional approach.

3.2.2 2D approach

In this subsection, we take the previously introduced three-dimensional sweep line algorithm and transform it into a two-dimensional sweep line algorithm. As discussed in Subsection 3.2.1, the relevant intersections are projected onto the ground plane. Instead of going through the trouble of sweeping a plane and projecting the relevant intersections, we could directly compute the edges on the two-dimensional Euclidean plane.

As explained in Subsection 3.2.1, sweep line algorithms maintain information about the intersection of the sweep line and the data structure of interest. In traditional sweep line algorithms the computation of the partial result above the sweep line is independent of everything below the sweep line. But in the case of Voronoi diagrams, sites that are yet to be encountered by the sweep line can have an influence on regions above the sweep line. For example, when the sweep line is at the location of the topmost vertex of the region of a site, it has not yet encountered the site itself. In Figure 3.3, we can see that not everything above the sweep line is fully known. This means that we do not yet have all the information that is needed to compute the vertices. To solve this, instead of maintaining the intersections of the Voronoi diagram and the sweep line, we maintain information about the part of the Voronoi diagram that cannot be influenced by sites below the sweep line.

We determine which part of the Voronoi diagram above the sweep line cannot be influenced by sites below the line by making use of the fact that edges and vertices of regions of neighboring sites are equidistant from those sites. That means that if the sweep line has not yet encountered a new site, a site above the sweep line either has a complete region or a partial region that is bounded by a line that is equidistant from the sweep line and the site. That line has the shape of a parabola. Each point on the parabola is equidistant from the site that it bounds and the sweep line, see Figure 3.4. As the sweep line moves, the shape of the parabola changes to reflect that the known size of a region changes. When the sweep line has encountered the first two sites, the intersection of the parabolas of those sites trace out the edge between them. This is due to the fact that the intersections are equidistant from both sites and the sweep line, see Figure 3.4. As the sweep line moves we are only interested in the parabolas of sites that have regions that are not yet fully computed. To be more concise, we are only interested in some parts of these parabolas, the beach arcs, which together form the beach line. The beach line is the function that for each x-coordinate passes through the lowest point of all parabolas at that x-coordinate.

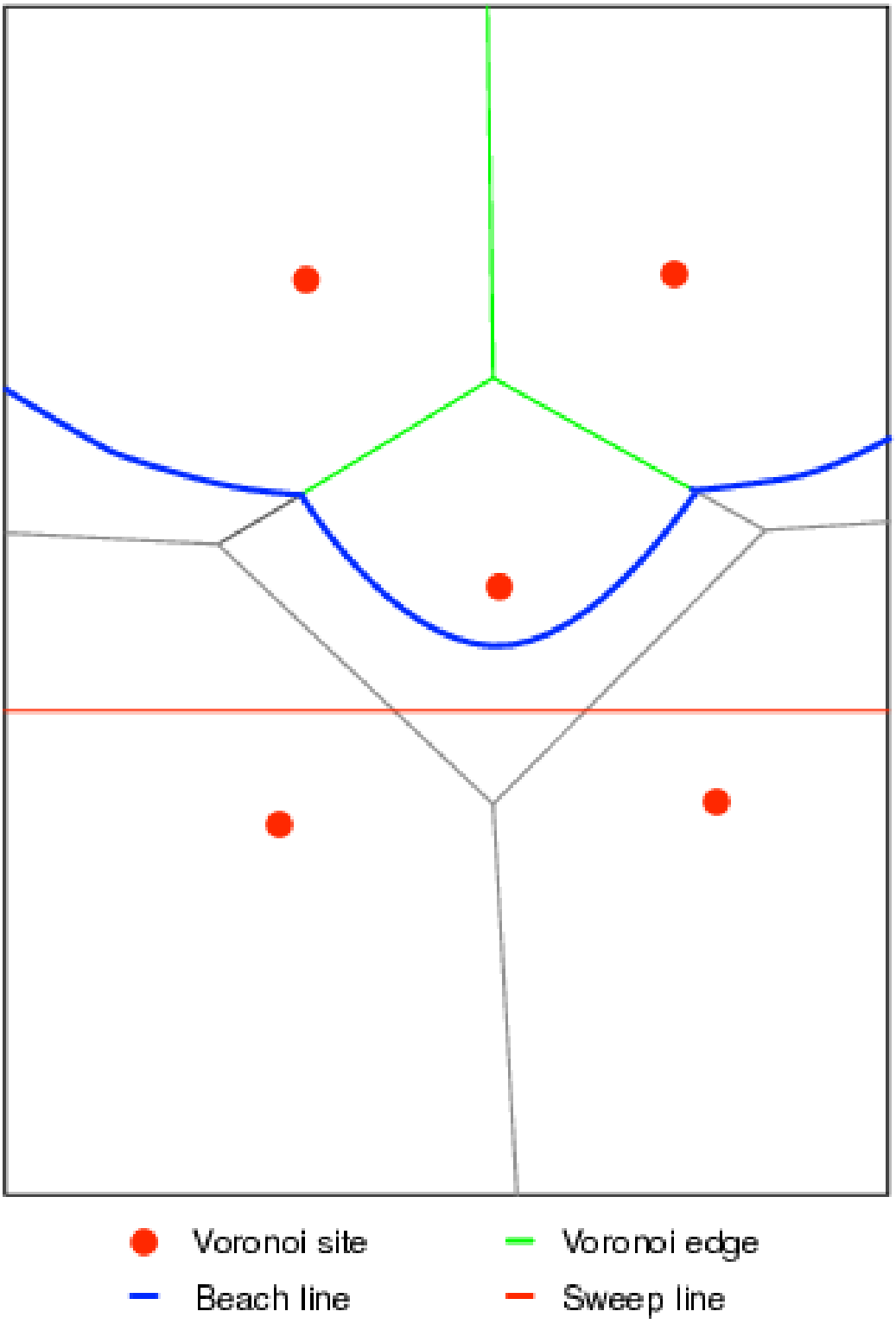

FIGURE 3.3: Fortune's sweep line algorithm

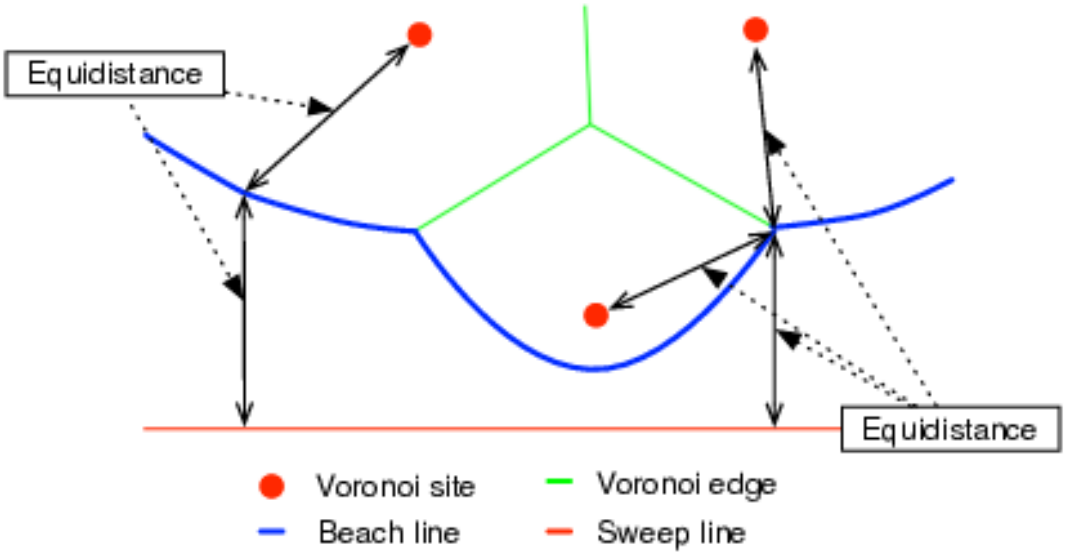

FIGURE 3.4: Beach line is equidistant from the sweep line and the Voronoi sites

Just like in the traditional sweep line algorithm, we maintain a structure while the sweep lines moves. But now we do not maintain the completed structure above the sweep line; instead we maintain the ever changing beach line. The only two events that are of importance when maintaining the beach line are when a parabolic arc enters the beach line and when one leaves the beach line. A new parabolic arc enters the beach line when the sweep line encounters a site. We call this event where a new site is encountered a site event. At the site event the parabola of the new site is at first a ray, starting in the site and perpendicular to the sweep line. This ray will intersect with the beach line. At this point of intersection the new parabolic arc will be inserted into the beach line, see Figure 3.5. The new parabola will intersect with at most two arcs on the beach line. A single site can contribute multiple times to the beach line as its parabola gets divided by new intersections with other arcs. As the sweep lines moves, arcs will grow and shrink, while their intersections with other arcs trace out the edges of the Voronoi diagram.

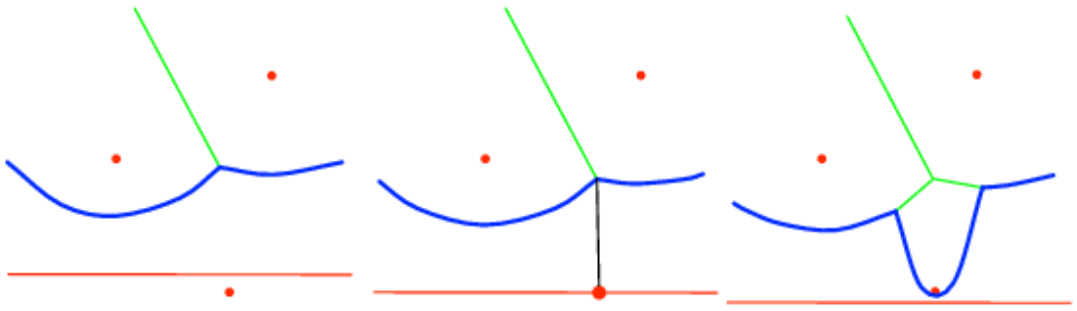

FIGURE 3.5: Site event: a new arc is added to the beach line (adapted from [14])

When a parabolic arc shrinks to a point, its neighboring parabolic arcs will meet. Figure 3.6 shows what happens when a parabolic arc shrinks to a point. This point where all three parabolic arcs meet is equidistant from three sites; therefore it is a vertex in the Voronoi diagram. This vertex is also the same distance from the sweep line as it is from the three sites. We can think of the vertex as the center point of a circle with a radius that is equal to the distance of the vertex to the sweep line or each of the three sites. When a parabolic arc disappears from the beach line, the sweep line passes through the lowest point of the circle. This is called a circle event.

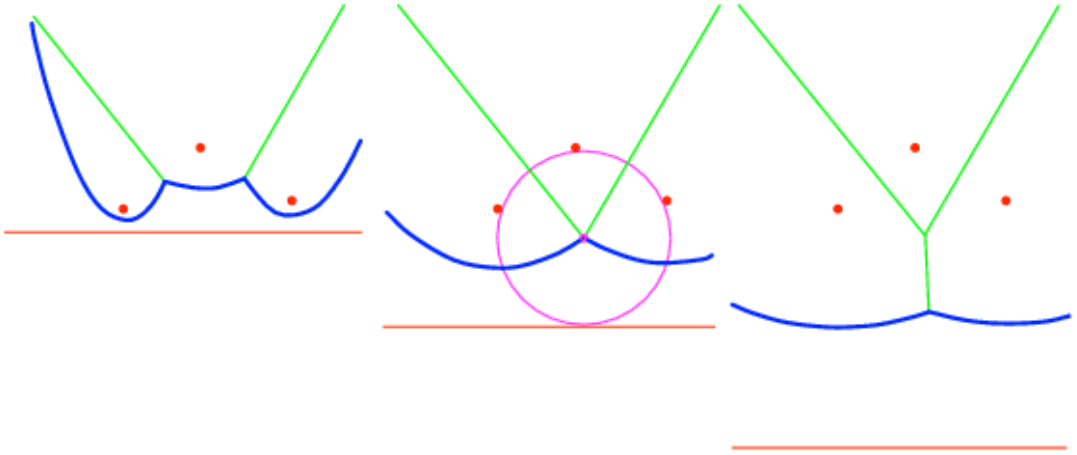

FIGURE 3.6: Circle event: an arc is removed from the beach line (adapted from [14])

There are only two events which change the structure of the beach line: a site event and a circle event. The only time a new parabolic arc is added to the beach line is at a site event. A parabolic arc is only removed from the beach line at a circle event. This means that the sweep line does not have to sweep the whole plane; instead, we can only sweep over those points that are of interest. In the next subsection we show how Fortune used this to his advantage in his algorithm.

3.2.3 Fortune's algorithm

In the previous two subsections we discussed the theory behind Fortune's algorithm. In this subsection, we introduce the actual algorithm and the data structures that are used. We closely follow the algorithms that are presented in Chapter 7 of [14]. To construct the Voronoi diagram we obviously need a way to store the current state of the Voronoi diagram. We also need a way to represent the beach line and a stucture to store the events that change the beach line, the event queue.

The part of the Voronoi diagram that is constructed is stored as a doubly-connected edge list. Each edge in a doubly-connected edge list stores two half-edges. The two half-edges are said to be twins of each other; if one half-edge starts in vertex v_1 and ends in vertex v_2, the twin starts in v_2 and ends in v_1. The half-edges enclosing a site form a counterclockwise cycle. Each half-edge stores a pointer to the next half-edge and a pointer to the previous half-edge. The half-edges also store the site they belong to and a pointer to their twin. Normally, doubly-connected edge lists can only deal with line segments. During the construction of the Voronoi diagram, not both the start and end point of all edges are known. Fortunately, this is no problem, because we can use the beach line to keep track of these unfinished edges. However, even in the completed Voronoi diagram, there are Voronoi regions that are not completely enclosed by edges. We add a bounding box to solve the problem of unfinished edges by connecting them to the sides of the bounding box. The bounding box is at least large enough to contain all the sites of the Voronoi diagram. This way we get a valid doubly-connected edge list.

To represent the beach line we use a balanced binary search tree. Each leaf in the tree corresponds to an arc on the beach line. The leafs are stored in an ordered manner: the leftmost leaf represents the leftmost arc, the next leaf represents the second leftmost arc, and so on. Actually, not the arcs are stored in the leafs but the sites that define the arcs. The internal nodes in the tree represent the intersection of two arcs. We store the intersections as ordered tuples of two sites $< s_i, s_j >$, where s_i can be used to compute the parabola on the left of the intersection and s_j can be used to compute the parabola on the right side of the intersection. When a new site is encountered by the sweep line, we have to look up which arc the parabola of the site will intersect with. Because the parabola at the moment of intersection of the sweep line and the site is a zero width parabola, we can look for the arc that is directly above the site. To find that arc in the beach line, we simply compare the x-coordinate of the site with the x-coordinate of the intersection that a node represents and traverse the tree accordingly. We do not directly store parabolas or arcs in the tree nodes, but we compute the intersections each time we do a look up by using the tuples of sites and the position of the sweep line.

Each internal node of the binary search tree not only stores the two sites that have intersecting arcs, but it also stores a pointer to a half-edge in the doubly-connected edge list. The pointer is to one of the half-edges of the edge that is being traced by the intersection that the node represents. Each leaf of the binary search tree stores a pointer to the node in the event queue that stores the circle event in which the arc corresponding to the leaf will disappear from the beach line. Since it is not known beforehand if and when an arc will disappear from the beach line, this pointer can be empty.

To store upcoming events we use a priority queue, the event queue. The priority queue is sorted on decreasing y-coordinate. Should two or more events have the same y-coordinate, we use their x-coordinates to determine which of them has a higher priority. For a site event we simply store the site itself. For circle events we store the lowest point of the circle and a pointer to the leaf in the binary search tree that corresponds to the arc that will disappear in the event.

In the previous subsections, we have discussed the concept behind the sweep line algorithm. We have introduced site events and circle events, the only two events in which the beach line changes. We know how to find site events; however, we have not discussed how circle events can be detected without sweeping the sweep line over the entire plane. We know from Subsection 3.2.2 that a circle event will take place when an arc shrinks to a point and its neighboring arcs meet. This means that when we want do detect circle events we have to look at three consecutive arcs. When a new site enters the beach line up to three of such triples can appear: one where the new site defines the left arc, one where it defines the middle arc and one where it defines the right arc. For the triples where a new arc appears on the left or right side we check if the two points of intersection converge to a single point. The triple where a new arc appears in the middle has two points of intersection that diverge so we do not have to check this triple. If the two points of intersection do converge for one of the other triples, the middle arc of the triple will disappear and we create a circle event. The middle arc will get a pointer to the circle event and the event is put into the event queue. Even if the two points of intersection converge to a single point, the circle event need not take place. For example, when a new site enters the beach line it can break up several existing triples.

We now have all the ingredients needed to give the full algorithm. However, we should note that after the event queue is empty we still have a beach line with pointers to half-edges of incomplete edges. As a doubly-connected edge list can not deal with incomplete edges, we still have to connect those to a bounding box. The algorithm is shown in Algorithm 1. The procedures that we define in this thesis and use in the algorithms are shown in capital letters. For example, Algorithm 1 uses Procedure HANDLESITEEVENT and Procedure HANDLECIRCLEEVENT. We use this convention for the rest of the thesis.

Input: A set $S = \{s_1, \ldots, s_n\}$ of sites in the Euclidean plane
Output: The Voronoi diagram for set S
begin
  Initialize priority queue Q with all site events;
  Initialize an empty binary search tree T;
  Initialize an empty doubly-connected edge list D;
  while Q *is not empty* **do**
    Remove the event with largest y-coordinate from Q;
    if *the event is a site event, occurring at site* s_i **then**
      HANDLESITEEVENT(s_i);
    else
      HANDLECIRCLEEVENT(y), where y is the leaf of T representing the arc that will disappear;
    end
  end
  The nodes still in T correspond to unfinished edges of the Voronoi diagram. Use a bounding box to connect all the unfinished edges to.
end

Algorithm 1: Fortune's sweep line algorithm for Voronoi diagrams

begin
  if T *is empty* **then**
    insert s_i into T, T now consists of a single leaf storing s_i;
  else
    Search in T for the arc a vertically above s_i. If the leaf representing a has a pointer to a circle event in Q, this circle event will not take place and is deleted from Q;
    Replace the leaf of T that represents a with a subtree that has three leaves. The middle leaf stores s_i, the other two leaves store s_j, the site that was originally stored at a. The two new nodes store the tuples $< s_j, s_i >$ and $< s_i, s_j >$, representing the new arc intersections in the beach line. Balance the tree if necessary;
    Create new half-edge records in the Voronoi diagram structure for the edge separating the region of s_i and the region of s_j. This edge will be traced out by the intersections of the new arcs;
    Check the triple of consecutive arcs where the new arc for s_i is either the right or left arc for converging arc intersections, i.e. check if the middle arc will disappear from the beach line. If the arc intersections converge, insert a circle event into Q and add pointers between the node in T and the node in Q;
  end
end

Procedure HANDLESITEEVENT(s_i)

begin
Delete the leaf y that represents the disappearing arc a from T. Update the node-tuples representing arc intersections. Re-balance T if necessary. Delete all circle events involving a from Q; This can be done by checking if the original neighboring arcs of a have a circle event associated with them;
Add the center of the circle causing the event as vertex record to the doubly-connected edge list. Create two half-edge records corresponding to the new edge that appears at the new arc intersection. Attach the vertex to all half-edges that end in that vertex. Do the same for the half-edges that start in that vertex;
Check the new triple of consecutive arcs that has the former left neighbor of a as its middle arc to see if the two arc intersections of the triple converge. If so, insert a new circle event into Q. Set the pointers between the new circle event in Q and the corresponding leaf of T. Do the same for the triple where the former right neighbor is the middle arc;
end

Procedure HANDLECIRCLEEVENT(s_i)

3.3 Fortune's algorithm extended to additively weighted power Voronoi diagrams

In this section, we explore the differences between the traditional Voronoi diagrams and additively weighted power (AWP) Voronoi diagrams. We will also show the changes that we made to Fortune's algorithm to create AWP Voronoi diagrams.

3.3.1 Additively weighted power Voronoi diagram definition

Additively weighted power Voronoi diagrams differ from traditional Voronoi diagrams by adding weights to the Voronoi sites and using a distance function that takes these weights into account. The distance function used is the power distance function. The power distance between Voronoi site O with weight w and point P is the square of the length of the line segment (PT) from P to a point T. T is the point on the circle with center point O and radius $\sqrt{w}$, such that PT is tangent to this circle. As seen in Figure 3.7, points P, T and O form a triangle with a right angle. Then it follows that the power distance between O and P is $\|P - O\|^2 - w$. This gives us the distance function for point p and site q with weight q_w as shown in Equation 3.1.

The weight added to a Voronoi site for AWP Voronoi diagrams has one restriction: it has to be greater than or equal to zero. This is explained by the fact that the weight can be seen as the radius of a circle squared. When we talk about the circle corresponding to a Voronoi site q, we mean the circle with center point q and radius $\sqrt{q_w}$. We also refer to this circle as "the circle of Voronoi site q".

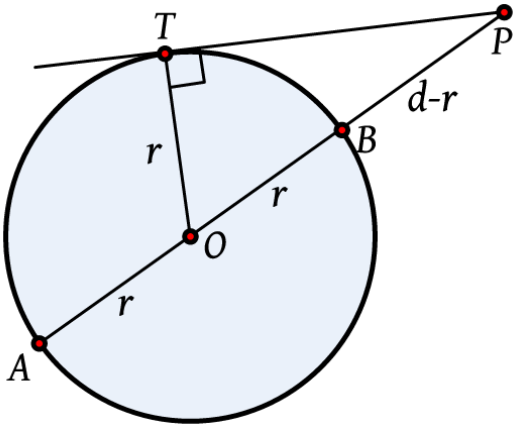

FIGURE 3.7: Power distance from point P to site O with weight $= r^2$ (reproduced from wikipedia.org)

$$dist(p, q, q_w) = (p_x - q_x)^2 + (p_y - q_y)^2 - q_w \tag{3.1}$$

The set of points at which tangents drawn to two circles have the same length is called the radical axis of those circles, see Figure 3.8. So the radical axis gives all points from which the power distance to the Voronoi sites of the corresponding circles is equal. The radical axis is a straight line perpendicular to the line segment between two Voronoi sites. This means that similar to Voronoi diagrams the Voronoi regions created using the power distance function are convex polygons. Figure 3.9 shows a Voronoi diagram and an AWP Voronoi diagram of the same set of sites; the difference that the distance function and the weights make is clearly visible.

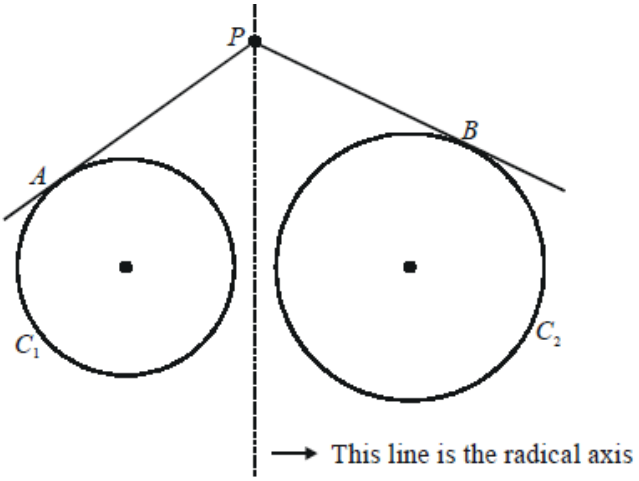

FIGURE 3.8: For any point p on the radical axis, tangents drawn from it to C_1 and C_2 are of equal length, i.e. $PA = PB$ (reproduced from cuemath.com)

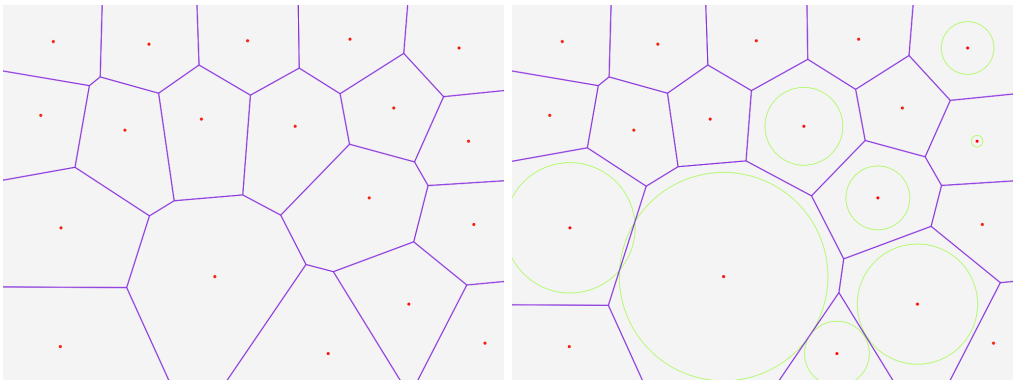

FIGURE 3.9: Voronoi diagram (left) and AWP Voronoi diagram (right) of the same set of sites

3.3.2 Sweep line, beach line and events

To extend Fortune's algorithm to create AWP Voronoi diagrams, we first need to identify how the Voronoi sites that the sweep line encounters influence the beach line. We also need to determine when a beach arc enters and disappears from the beach line. After that, we have to determine the order of the events and the changes needed to make AWP Voronoi diagrams.

3.3.2.1 Site events

When moving the sweep line over a set of weighted sites, there are two places where a Voronoi site could have an effect on the beach line: at the top of its circle and at the site itself. When the sweep line touches the top of the circle, the power distance from the sweep line to the Voronoi site is zero. When this happens we originally thought that this would introduce a new beach arc into the beach line. This turned out to be wrong. As shown in Figure 3.10, Voronoi sites that will be encountered later by the sweep line than the top of the circle of a Voronoi site with a large weight can influence the beach line before the big weighted Voronoi site influences it. This means that the order of site events remains the same as in the original algorithm. So we insert a new beach arc into the beach line when the site itself is encountered.

3.3.2.2 Circle events

The second event that we need to consider is the circle event. In Fortune's algorithm a circle event occurs when three Voronoi sites are on the same circle. When using weighted Voronoi sites and the power distance function, we can not use the same method. What we can use is the radical center of three circles corresponding to three Voronoi sites. The

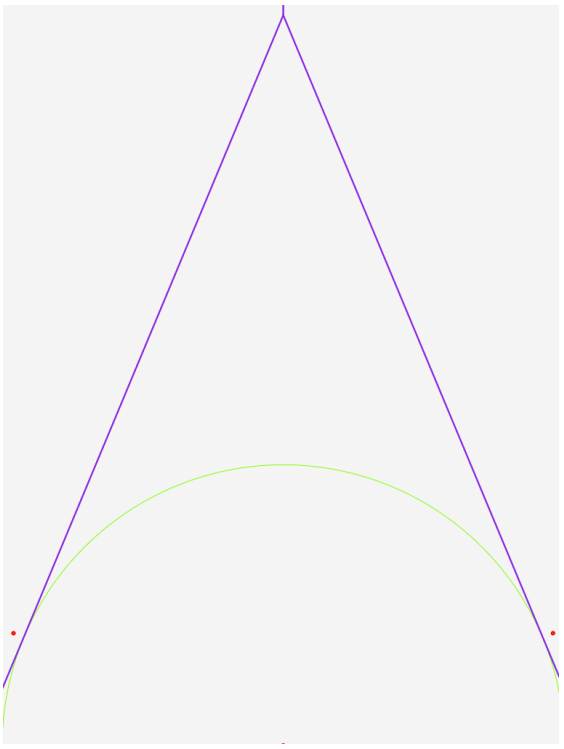

FIGURE 3.10: An AWP Voronoi diagram of three sites, one site with a large weight and its corresponding circle and two sites that lie lower than the top of that circle

radical center can easily be found by intersecting the radical axis of each pair of the three circles, see Figure 3.11. The circle with its center at the radical center and orthogonal to all three circles, is the circle for our circle event for AWP Voronoi diagrams. When the sweep line encounters the bottom of this orthogonal circle, an arc disappears from the beach line.

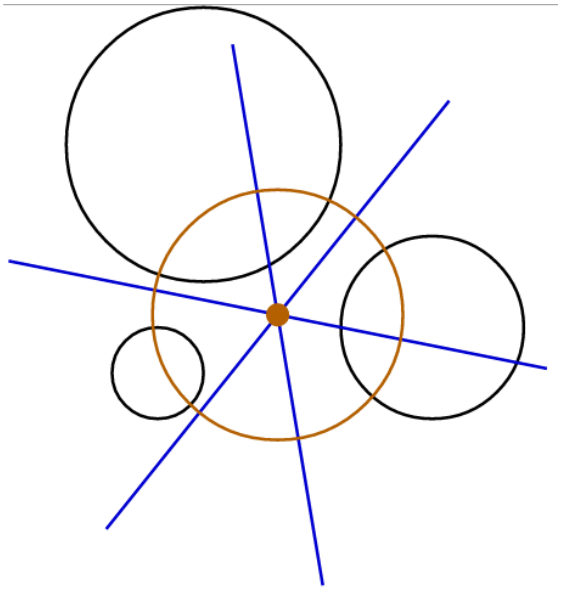

FIGURE 3.11: The radical center and the orthogonal circle of three circles corresponding to weighted Voronoi sites (reproduced from wikipedia.org)

3.3.2.3 Sweep line and beach line

The beach line used in our algorithm is largely similar to what is used in Fortune's algorithm. Instead of using the Euclidean distance function to calculate the arc intersections, we use the power distance function. Later on in this section we will show that we need to add one more change.

The sweep line we use in our algorithm differs greatly from the one used in Fortune's algorithm. In Fortune's algorithm the sweep line is a straight line. We found that this will not work for AWP Voronoi diagrams. When the sweep line encounters the top of a circle of a Voronoi site, it does not introduce a new beach arc into the beach line. But it does have an effect when the sweep line enters a circle. In Figure 3.12, we see three parabolas (blue, red, purple), their corresponding sites, and the sweep line (black). We also see one Voronoi site with a large weight that the sweep line has not yet encountered the site event for. We can clearly see that parts of the beach line lie within the circle of this site. This is not correct. The beach arcs describe the area of a Voronoi region that is completely known, but in this case the Voronoi region should not extend to inside the circle.

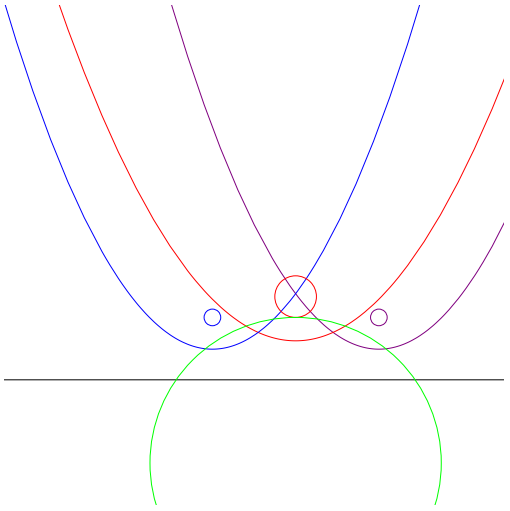

FIGURE 3.12: Beach line and sweep line with a site with a large weight that is yet to be encountered

Our solution to the problem of the beach line entering a circle of a site is the introduction of a new event and a small change to the beach line. The location of the new event is where the sweep line encounters the top of the circle of a site; we call this a "top site circle event". We look for the beach arc that lies directly above the site. We mark this arc and maybe its neighbors. This mark indicates that the arc is influenced by the site. When computing intersections of beach arcs we use this information. We found that the marked arc should not be at equal power distance from the sweep line, but from the site it is marked with. When one of two arcs is marked, we calculate the bisector of the site

corresponding to the marked arc and the site the arc is marked with. We also calculate the bisector of the neighbor site and the site the other arc is marked with. We then intersect the two bisectors, and check if the power distance from the intersection to the sweep line is smaller than to the sites. If it is closer to the sweep line, we use the normal intersection calculation. Otherwise, we use the intersection of the bisectors. Figure 3.13 shows how this would look by approximation. In Subsection 3.3.2.5 we will go into the details of how we mark beach arcs.

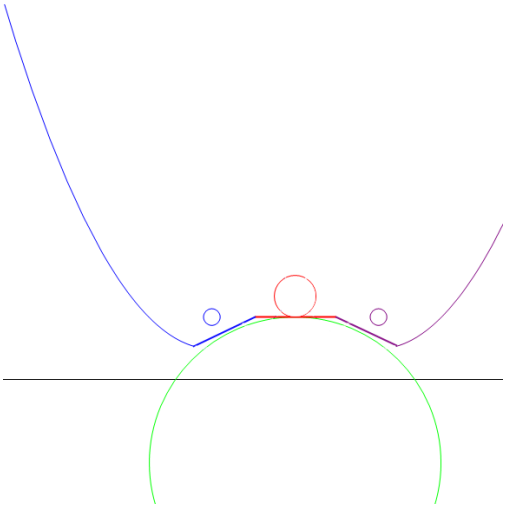

FIGURE 3.13: Updated beach line with bisectors

There is another problem that the weighted sites introduce: when the site event of a site with a large weight has not happened yet, circle events that have an event location that lie within the circle of the site do happen. This means that we remove an arc from the beach line, create a new vertex and a new edge. None of these actions should happen. To solve this, we change the sweep line from just a straight line to a line that incorporates the circles of sites that are yet to be encountered. In Figure 3.14 you can see what this would look like. We use this new sweep line to check whether circle events may happen, based on whether a circle event location is above or below the sweep line.

We remove a circle from the sweep line when its corresponding Voronoi site has entered the beach line. This means that on a site event we can remove the circle from the sweep line. When the sweep line has not fully passed through the circle of a site, it will still produce a useful parabola that we can use to calculate the beach arc intersections. Figure 3.15 shows two examples. From these examples we can see that the parabolas are not correct below the sweep line. As we make no claims about anything below the sweep line, this is not a problem.

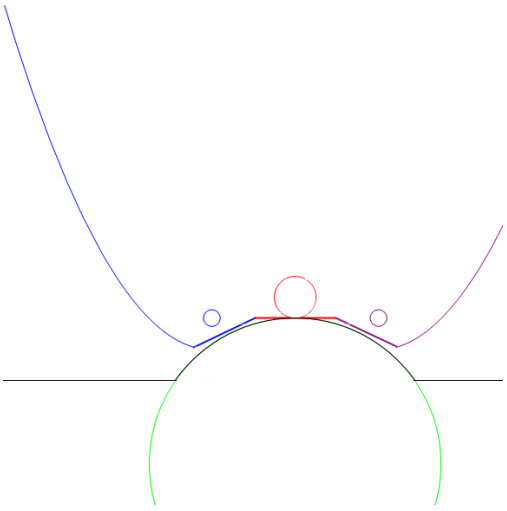

FIGURE 3.14: Updated sweep line with a site that is yet to be encountered

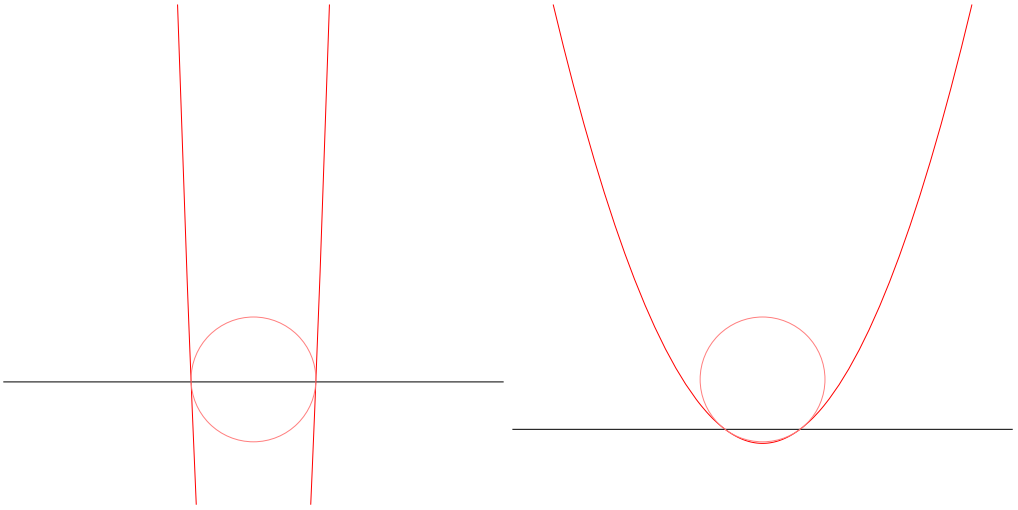

FIGURE 3.15: The parabolas of two Voronoi sites when the sweep line is below the site, but not below the corresponding circle

3.3.2.4 Top site circle event

Top site circle (TSC) events happen when the sweep line encounters the top of a circle corresponding to a Voronoi site. Therefore, we can order the events by the y-coordinate of the top of the circles. If two events have the same y-coordinate we use the x-coordinate of the Voronoi sites.

3.3.2.5 Beach arcs

As described in Subsection 3.3.2.3, we had to make a small change to the beach arcs. In Fortune's algorithm a beach arc consists of a Voronoi site, a left and right neighbor, and possibly the reference to a circle event. We add two possible references to Voronoi

sites that influence a beach arc. One reference to the site that is used to calculate the left intersection and one to the site that is used to calculate the right intersection. A beach arc can be infinitely large and might be influenced by many sites, but we are only interested in the leftmost and rightmost that lie below the arc. So just keeping a reference to the leftmost and rightmost suffices. When the beach arc is small or the Voronoi site that influences it has a large weight, both the left and right reference can point to the same Voronoi site.

When a TSC event happens, we find the arc that lies right above the site that the event belongs to. If the arc is not marked yet, we set both the left and right reference. We then check for all neighbors if they are also influenced by the TSC event. If they are, we also mark them. To check if an arc is influenced by a Voronoi site, we check if the arc is (fully or partially) above the circle. We also check the neighbors if this is the case.

When an arc is already marked with a reference to a site, and another TSC event also influences it, we check whether the new site is leftmost or rightmost. If it is leftmost or rightmost we update the corresponding reference. We also check the corresponding neighbor arc.

When a site event happens and a new arc is introduced into the beach line, we check if the neighbors of the new arc had references to the site belonging to the new arc. We remove all of those references. We also check if the updated arcs are above the circle of a site that is still in the sweep line and set the correct references.

One special case we have to take into account is that the first TSC event is always before a site event and we need to handle that correctly. If a TSC event happens and the beach line is empty, we postpone it until we have at least one beach arc in the beach line.

When calculating the beach arc intersections, we use the references to calculate the intersections. If we calculate the left intersection of an arc, we check if the arc or its left neighbor have a reference to a site. For the current arc we check the left reference, and for the left neighbor we check the right reference. If there is only one reference, we calculate the intersection of the bisector of the arc site and the reference site and the bisector of the left neighbor site and the reference site. If both have a reference, we check the intersection of the bisectors of the arc site and the corresponding reference site. We then check if the found intersection is closer to the sweep line or to the reference site(s). If it is closer to the sweep line we fall back on the traditional arc intersection by calculating the intersection of the parabolas.

3.4 Algorithm limitations

Due to the way we order events, a site should not be located in the circle corresponding to another site. If a site lies within the circle of another site, the beach line intersections we calculate would sometimes give incorrect results. Overlap of circles is not a problem.

3.5 The algorithm

Putting all our changes together with Fortune's algorithm, we get the algorithm as shown in Algorithm 2.

Data: A set $S = \{s_1, ..., s_n\}$ of weighted sites with no site in the circle of another site
A convex polygon P that bounds the sites in S
Result: The bounded additively weighted power Voronoi diagram of S in a doubly-connected edge list D

```
begin
    Initialize site event queue SQ with all site events from S;
    Initialize TSC event queue TQ with all TSC events from S;
    Initialize an empty circle event queue CQ;
    Initialize an empty beach line B;
    Initialize an empty sweep line L;
    Initialize an empty doubly-connected edge list D;
    Pop site event from SQ and insert the corresponding site into B;
    se ← pop site event from SQ;
    te ← pop TSC event from TQ;
    while true do
        ce ← FINDNEXTCIRCLEEVENT(CQ, L);
        if se == null and ce == null then
            break;
        end
        else if se < te and se < ce then
            HANDLESITEEVENT(se, B, CQ, D, L);
            se ← pop site event from SQ;
        end
        else if te < se and te < ce then
            HANDLETSCEVENT(te, B, L);
            te ← pop TSC event from TQ;
        end
        else if ce < se and ce < te then
            HANDLECIRCLEEVENT(ce, B, CQ, D);
        end
    end
    Intersect D with P to create the bounded Voronoi diagram;
end
```

Algorithm 2: Sweep line algorithm for AWP Voronoi diagrams

Data: Site event se
Beach line B
Circle event queue CQ
Doubly-connected edge list D
Sweep line L
Result: D updated with new edges
CQ updated with possible new circle events
begin
    $s \leftarrow$ site corresponding to se;
    Remove s from L;
    Update all arcs that reference s;
    $a \leftarrow$ beach arc in B vertically above s;
    if *a has associated circle event* **then**
        remove circle event from CQ;
    end
    $sa \leftarrow$ the site associated with a;
    Split a into two arcs al and ar;
    Create new beach arc an with site s;
    Insert an between al and ar;
    Update B;
    Create edge e between s and sa;
    Insert e into D;
    Check for a circle event for al and its two neighbors;
    Check for a circle event for ar and its two neighbors;
    Insert found circle events into CQ;
end

Procedure HANDLESITEEVENT(se, B, CQ, D, L)

Data: Circle event ce
Beach line B
Circle event queue CQ
Doubly-connected edge list D
Result: D updated with new edges
CQ updated with possible new circle events
begin
    remove ce from CQ;
    $s \leftarrow$ site corresponding to ce;
    $a \leftarrow$ beach arc associated with ce;
    $e \leftarrow$ edge associated with a;
    $al \leftarrow$ left neighbor of a;
    $ar \leftarrow$ right neighbor of a;
    $v \leftarrow$ center point of circle event ce;
    End e in vertex v;
    Connect al and ar so that they are direct neighbors;
    Create edge f between the sites of al and ar;
    Start f with vertex v;
    Add f to D;
    Update B;
    Check for a circle event for al and its two neighbors;
    Check for a circle event for ar and its two neighbors;
    Update site references for al;
    Update site references for ar;
    Insert found circle events into CQ;
end

Procedure HANDLECIRCLEEVENT(ce, B, CQ, D)

```
Data: TSC event te
Beach line B
Sweep line L
Result: Arcs in B updated to reflect new site references
begin
    s ← site belonging to te;
    a ← beach arc in B vertically above s;
    sl ← left site reference of a;
    sr ← right site reference of a;
    if sl == null or s.x < sl.x then
        Set left reference of a to s;
        if intersection of a and left neighbor above circle s then
            SETARCSITEREFERENCE(e, a.leftNeighbor, L);
        end
    end
    if sr == null or s.x > sr.x then
        Set right reference of a to s;
        if intersection of a and right neighbor above circle s then
            SETARCSITEREFERENCE(e, a.rightNeighbor, L);
        end
    end
end
```

Procedure HANDLETSCEVENT(te, B, L)

```
Data: Circle event queue CQ, events are ordered by their y-coordinate
Sweep line L
Result: The next circle event that may happen
begin
    foreach ce ∈ CQ do
        if ce above y-coordinate L then
            neverinside ← true;
            foreach s that is on L do
                if ce event location inside circle of s then
                    neverinside ← false;
                    break;
                end
            end
            if neverinside then
                return ce;
            end
        end
        else
            break;
        end
    end
    return null;
end
```

Procedure FINDNEXTCIRCLEEVENT(CQ, L)

```
Data: TSC event te
Beach arc a
Beach line L
Result: Updated arc with left and right site reference
begin
    s ← site belonging to te;
    sl ← left site reference of a;
    sr ← right site reference of a;
    if sl == null or s.x < sl.x then
        Set left reference of a to s;
        if intersection of a and left neighbor above circle s then
            SETARCSITEREFERENCE(e, a.leftNeighbor, L);
        end
    end
    if sr == null or s.x > sr.x then
        Set right reference of a to s;
        if intersection of a and right neighbor above circle s then
            SETARCSITEREFERENCE(e, a.rightNeighbor, L);
        end
    end
end
```

Procedure SETARCSITEREFERENCE(te, a, L)

Chapter 4

Voronoi treemaps

In Chapter 3, we have shown how to compute additively weighted power Voronoi diagrams. In this chapter we will show how we use those diagrams to create a Voronoi treemap. First, we introduce centroidal Voronoi diagrams and the changes needed for Voronoi treemaps. Then we show how we combine it all to create a Voronoi treemap.

4.1 Centroidal Voronoi diagrams

Centroidal Voronoi diagrams, more often referred to as centroidal Voronoi tessellations, are Voronoi diagrams where the sites are at the center of weight, the centroid, of their Voronoi region. Centroidal Voronoi diagrams have as property that the plane is divided in such a manner that all Voronoi regions have approximately the same area. Centroidal Voronoi diagrams are often used to study the territorial behaviour of animals. In [31], the authors give a concise overview of centroidal Voronoi diagrams, their uses and the algorithms to create them.

To compute a centroidal Voronoi diagram we use Lloyd's algorithm as described in [31]. Lloyd's algorithm uses iterative relaxation of the Voronoi diagram. First, the original Voronoi algorithm is used to compute the Voronoi regions of a set of sites. Second, each site is placed at the center of weight of its region. To compute the center of weight for each region we compute the centroid C, where $C = (C_x, C_y)$, using equation 4.2 to compute C_x and equation 4.3 to compute C_y. Both of these equations use the area A of the Voronoi region, which is given by equation 4.1. All three equations assume that the Voronoi region is closed, which is no problem because we close all open Voronoi regions with a bounding box. The vertices $(x_0, y_0), (x_1, y_1), ..., (x_{n-1}, y_{n-1})$ of the region are ordered counterclockwise and the vertex (x_n, y_n) is the same as (x_0, y_0). We use the

center of weight of each region as the new location for each of the sites to compute the new Voronoi diagram. We repeat this till we come to a stable point, where each site is already at the center of weight of its Voronoi region. Figure 4.1 shows the results of Lloyd's algorithm at several intervals. Usually, we allow for a small margin of error, where each site is at a small distance from the center of weight. This ensures that the algorithm finishes in a timely manner. The algorithm is given in Algorithm 3.

$$A = \frac{1}{2}\sum_{i=0}^{n-1}(x_i\ y_{i+1} - x_{i+1}\ y_i) \tag{4.1}$$

$$C_x = \frac{1}{6A}\sum_{i=0}^{n-1}(x_i + x_{i+1})(x_i\ y_{i+1} - x_{i+1}\ y_i) \tag{4.2}$$

$$C_y = \frac{1}{6A}\sum_{i=0}^{n-1}(y_i + y_{i+1})(x_i\ y_{i+1} - x_{i+1}\ y_i) \tag{4.3}$$

```
Data: A set S = {s_1, ..., s_n} of sites
The minimal distance d that a site needs to move
Result: The centroidal Voronoi diagram of S in a doubly-connected edge list D
begin
    D ← CreateVoronoiDiagram(S);
    notdone ← true;
    while notdone do
        notdone ← false;
        foreach s in S do
            a ← Voronoi region of s in D;
            cow ← center of weight of a;
            if distance(s, cow) > d then
                notdone ← true;
            end
            Set location of s to cow;
        end
        D ← CreateVoronoiDiagram(S);
    end
    return D;
end
```

Algorithm 3: Lloyd's algorithm for computing centroidal Voronoi diagrams

4.2 Additively weighted power centroidal Voronoi diagrams

To create Voronoi diagrams, we have to divide an area into subdivisions that have a size corresponding to a metric. By using AWP Voronoi diagrams alone we can not create such subdivisions. However, if we combine AWP Voronoi diagrams with centroidal Voronoi

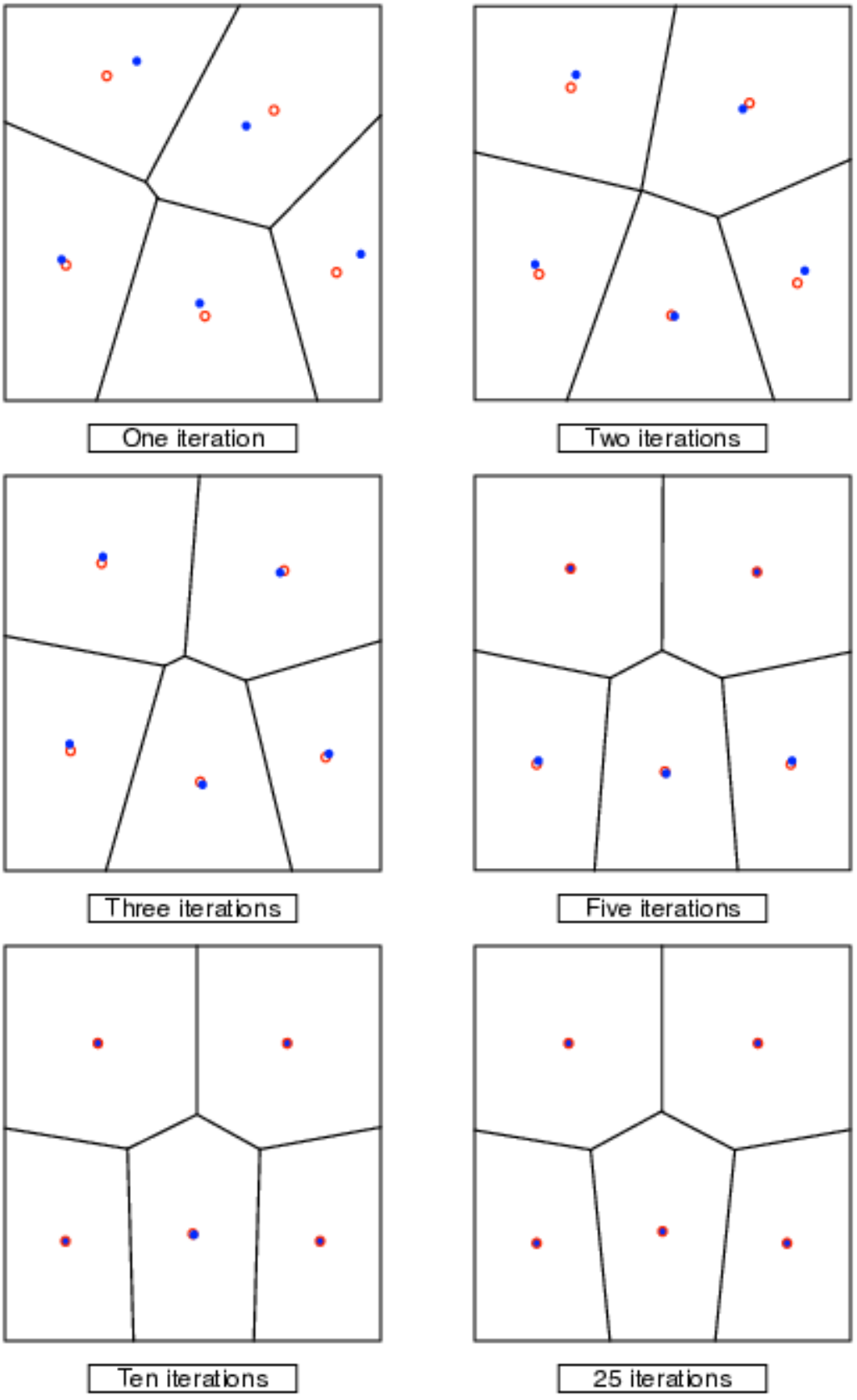

FIGURE 4.1: Several iterations of Lloyd's algorithm

diagrams and not only change the location of the site but also its weight, we can create the desired subdivisions. We also change the condition on which the algorithm stops. We check the size of the subdivisions and when all subdivisions are within a small margin of error of their desired size we stop. This algorithm was first described in [2] and later extended in [12]. Our algorithm very closely resembles the algorithm as described in [12], but due to our different data structures it differs slightly.

In [12], Nocaj and Brandes identify two distinct steps in each iteration of the algorithm. In the first step they change both the position and the weight of a Voronoi site. In the second step they only change the weights. While both steps change the weight, the way they do it is different.

In the first step, they start by moving the site to the centroid of its Voronoi region. Then they change the weight, but not in relation to the desired size of the region. When moving a site with a large weight, its corresponding circle might accidentally overlap with a neighboring site, and as per Section 3.4 this is not allowed. They prevent this by changing the weight after the site has been moved. The weight is changed if the radius of the circle is larger than the Euclidean distance to the nearest edge. Figure 4.2 shows that if site s is moved and its original weight (red) is kept, then its circle will overlap with site v. By changing the weight such that the circle lies within the Voronoi region of s (blue), this is prevented.

The second step is used to influence the size of the region belonging to the site. Considering the limitations of our algorithm, as described in Section 3.4, the circle corresponding with the site should never be so large that it overlaps with another site. Therefore, we limit the weight in such a way that the circle cannot overlap with the neighbors of the site. We have to use the neighbors in the standard Voronoi diagram, because a site might be a neighbor of a site in the ordinary Voronoi diagram, without being its neighbor in the AWP Voronoi diagram, see Figure 4.3.

Taking all this into consideration we come to Algorithm 4 to create a single layer of a Voronoi treemap.

4.3 Recursively building a Voronoi treemap

We now have all the ingredients needed to create a Voronoi treemap. We know how to compute the individual subdivisions, so all that is left is to combine them with a hierarchical graph. Algorithm 5 shows how this is done. We start with creating the first level of the Voronoi treemap by providing the algorithm with the hierarchical graph and a convex polygon that will bound the Voronoi treemap. We divide the convex polygon

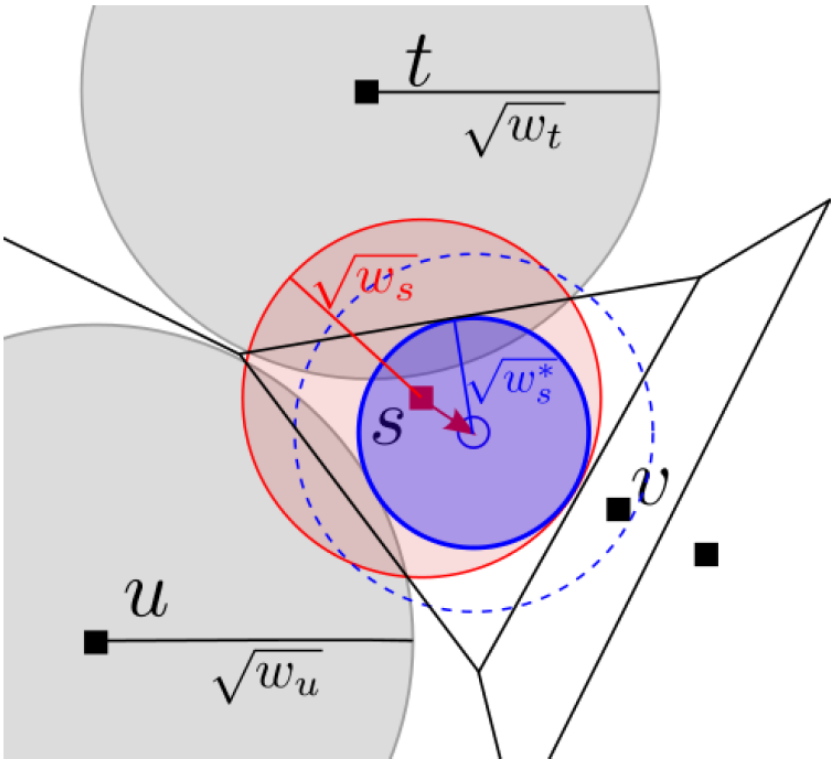

FIGURE 4.2: Moving site s without changing its original weight would mean that site v lies within the circle belonging to s. (reproduced from [12])

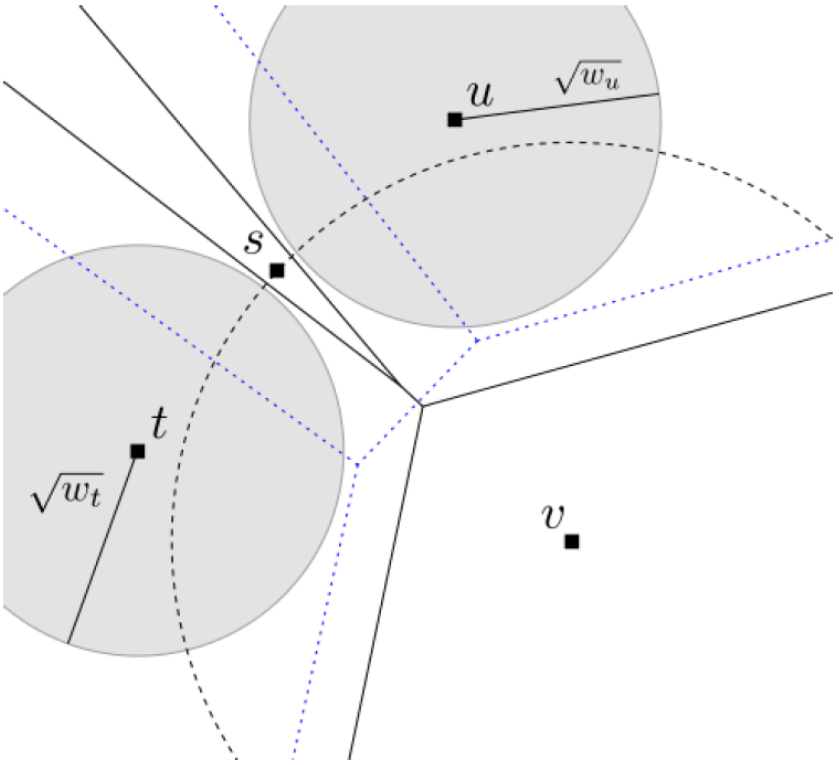

FIGURE 4.3: If we increase the weight of site v and only take into account its neighbors in the AWP Voronoi diagram, then the circle belonging to v might overlap with site s. This is prevented by using the neighbors in the standard Voronoi diagram (dotted blue line). (reproduced from [12])

Data: A set $S = \{s_1, ..., s_n\}$ of weighted sites with no site in a circle corresponding to another site
Each site in S has an associated desired area
Each site in S has a starting weight of 1
A convex polygon P that bounds S
i_{max} max number of iterations
$E_{threshold}$ error threshold
Result: A doubly-connected edge list D bounded by P
begin
  $D \leftarrow$ initialize empty doubly-connected edge list;
  for $i \leftarrow 1$ **to** i_{max} **do**
    $D \leftarrow$ create an AWP Voronoi diagram of S bounded by P;
    ADAPTPOSITIONSANDWEIGHTS(D, S);
    $D \leftarrow$ create an AWP Voronoi diagram of S bounded by P;
    $totalError \leftarrow$ ADAPTWEIGHTS(D, S, P);
    $error \leftarrow \frac{totalError}{2 \times Area(P)}$;
    if $error < E_{threshold}$ **then**
      return D;
    end
  end
  return D;
end

Algorithm 4: Compute Voronoi treemap (single layer)

Data: A set $S = \{s_1, ..., s_n\}$ of weighted sites with no site in a circle corresponding to another site
Each site in S has an associated desired area
A bounded doubly-connected edge list D
Result: All sites in S updated with a new location and weight
begin
  foreach $s \in S$ **do**
    $a \leftarrow$ the Voronoi region of s in D;
    $cow \leftarrow$ the center of weight of a;
    Set location s to cow;
    $minBorderDist \leftarrow$ min distance from s to edges in D;
    $weight \leftarrow$ weight of s;
    $newWeight \leftarrow min(\text{weight}, minBorderDist^2)$;
    Set weight of s to $newWeight$;
  end
end

Procedure ADAPTPOSITIONSANDWEIGHTS(D, S)

Data: A set $S = \{s_1, ..., s_n\}$ of weighted sites with no site in a circle corresponding to another site
Each site in S has an associated desired area
A bounded doubly-connected edge list D
A convex polygon P
Result: All sites in S with updated weight
The sum of errors of the desired size of each site versus the actual size
begin
    `// Please note we create a normal Voronoi diagram`
    $D' \leftarrow$ create a Voronoi diagram of S bounded by P;
    $totalError \leftarrow 0$;
    foreach $s \in S$ **do**
        $nn \leftarrow$ nearest neighbor of s in D';
        $a_{current} \leftarrow$ current size of the Voronoi region of s in D;
        $a_{desired} \leftarrow$ desired size of the Voronoi region of s;
        $totalError \leftarrow totalError + abs(a_{current} - a_{desired})$;
        $factor \leftarrow \frac{a_{desired}}{a_{current}}$;
        $w \leftarrow$ current weight of s;
        $newWeight \leftarrow min(w \times factor, distance(s, nn)^2)$;
        Set weight of s to $max(newWeight, 1)$;
    end
    return $totalError$;
end

Procedure ADAPTWEIGHTS(D, S, P)

into Voronoi regions, one for each of the children of the root node in the hierarchical data. These Voronoi regions are then divided for each of their children. We keep doing this till we have created a Voronoi region for every node in the hierarchical data.

Data: A hierarchical graph H with nodes with associated metric m
A convex polygon P
Result: A Voronoi treemap as a hierarchical graph of doubly-connected edge lists
begin
  $r \leftarrow$ get root node of H;
  $N \leftarrow$ child nodes of r in H;
  $S \leftarrow$ empty collection of Voronoi sites;
  $a \leftarrow$ area size of P;
  foreach $n \in N$ **do**
    $s \leftarrow$ create a new Voronoi site that lies in P;
    $s_{desiredarea} \leftarrow a \times \frac{n_m}{r_m}$;
    Add s to S;
  end
  $D \leftarrow$ compute the single layer of S bounded by P;
  foreach $n \in N$ **do**
    if *n has children in H* **then**
      $s \leftarrow$ the site in S that corresponds to n;
      $P' \leftarrow$ get the polygon of s in D;
      $H' \leftarrow$ get sub graph with root n from H;
      $D' \leftarrow$ CreateVoronoiTreemap(P', H');
      Add D' to D as child;
    end
  end
  return D;
end

Algorithm 5: Voronoi treemap creation

Chapter 5

Stable Voronoi treemap algorithm

Up until now the Voronoi treemaps do not take the order and placement of the Voronoi sites into account. This means that each Voronoi treemap will be completely different, even when visualizing the same data. When we visualize the same data multiple times we can not easily relate the visualizations to each other. This also holds for visualizing data that changes over time. Each visualization is completely different while the underlying data might not differ that much.

To solve this, we introduce several algorithms that ensure that visualizations that do not differ in underlying data do not differ visually. We also show that these algorithms can be used to visualize data that changes over time.

5.1 Voronoi site ordering

One of the reasons Voronoi treemaps differ from each other is that there is no well-defined order of Voronoi sites. Each time data is used to create a Voronoi treemap, sites are created in a random order.

We propose to enforce a strict order on the creation of Voronoi sites. The order we propose to use is the order of the underlying data. If one data point is presented later in the data than another, then its Voronoi site will be created later. This ensures that the order of Voronoi sites is equal to the order of the underlying data. This means that the algorithms need to be provided with a data set that has a well-defined order. If the data is not ordered, the visualization will not be either. If we visualize data that changes over time but we do keep the same order, then the visualizations will also keep their order.

5.2 Hilbert curve Voronoi site placement

The second problem we face when trying to create a stable Voronoi diagram algorithm is the placement of Voronoi sites. Even if we enforce a strict order on the creation of Voronoi sites, but then place them at random locations, we still get completely random Voronoi treemaps.

Order is enforced on the data; therefore the creation of Voronoi sites can be seen as placing all sites in a single line. By using a space-filling curve, we can place this line over the polygon used for our visualization and place the Voronoi sites accordingly. This ensures that we place them at a predictable location. The space-filling curve we use is the Hilbert curve. Figure 5.1 shows Hilbert curves of order 1, 2 and 3. Hilbert curves are constructed by visiting every cell in a square grid. The order in which each cell is visited is based on a simple recursive algorithm that is described in Chapter 14 of [32]. When we visit a cell we add the center point of the cell to a list of points on the Hilbert curve. This list of points is what we are interested in.

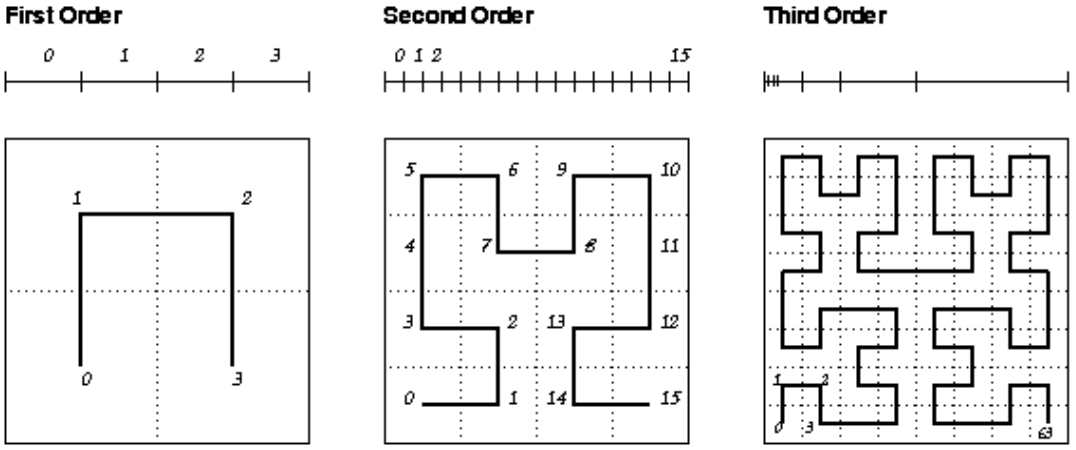

FIGURE 5.1: Hilbert curves of order 1, 2, and 3 (reproduced from opengl.org)

When creating Voronoi sites we have to place them within the convex polygon that we are dividing. We place a Hilbert curve over the polygon, but a Hilbert curve, by definition, lies in a perfect square and the polygon most likely will not be a square at all. We solve this by scaling and moving the Hilbert curve, so that it covers the complete polygon. We move the Hilbert curve until the lowest point of the polygon lines up with the lowest edge of the Hilbert square. We also position the left edge of the Hilbert square such that it lines up with the leftmost point of the polygon. We then scale the Hilbert square to cover the complete polygon. By intersecting the points on the Hilbert curve with the polygon, we get the set of points we can use to create Voronoi sites, see Figure 5.2. When creating Voronoi sites, we pick points from this set at regular

intervals. We pick them in the order that they occur on the Hilbert curve, see Figure 5.3 and Algorithm 6.

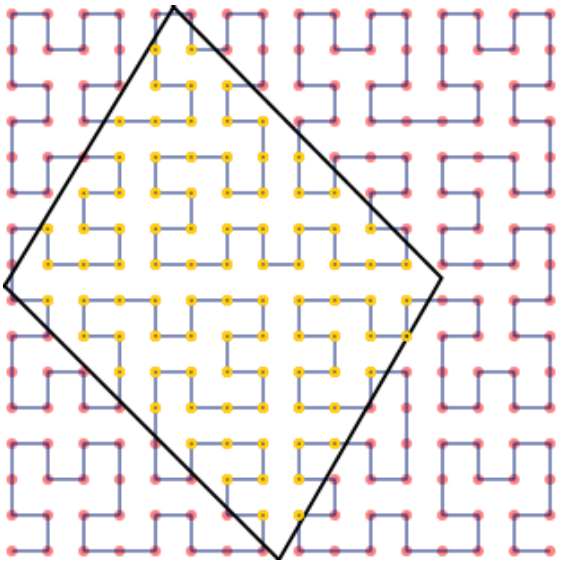

FIGURE 5.2: A Hilbert curve intersected with a polygon

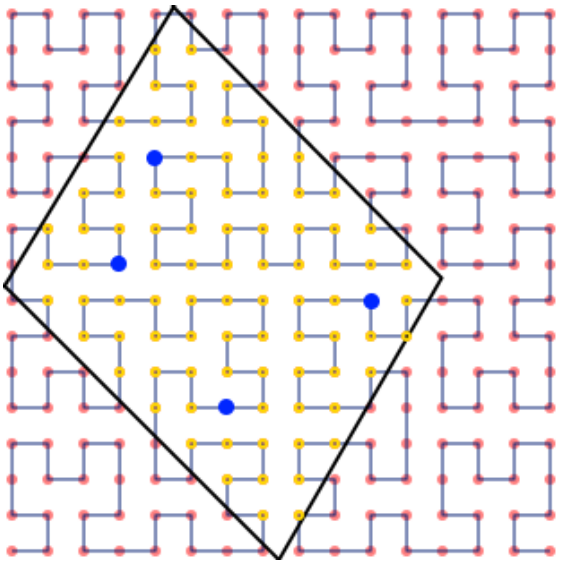

FIGURE 5.3: A Hilbert curve intersected with a polygon and four Voronoi sites placed at equal intervals

In our examples we use Hilbert curves of at most order 4, but in our actual implementation we use a grid of $2^8 \times 2^8$. This gives us a greater number of points in case the shape of the polygon is very elongated and would otherwise intersect with very few points.

Data: A set $P = \{p_1, ..., p_i\}$, consisting of all points in the intersection of the Hilbert curve and some convex polygon
A set $N = \{n_1, ..., n_j\}$ of nodes
Result: A set $S = \{s_1, ..., s_j\}$ of Voronoi sites
begin
  $i \leftarrow$ number of points in P;
  $j \leftarrow$ number of nodes in N;
  $stepsize \leftarrow \frac{i}{j}$;
  $index \leftarrow \frac{stepsize}{2}$;
  foreach $n \in N$ **do**
    Create Voronoi site s at p_{index};
    $index \leftarrow index + stepsize$;
    Add s to S;
  end
  return S;
end

Algorithm 6: Hilbert curve Voronoi site creation

5.3 Variable Hilbert curve Voronoi site placement

We realized that using a Hilbert curve to place Voronoi sites could be extended to take the desired area of the region of the sites into account. Instead of placing the sites at equal intervals, we could scale them according to the desired size of their region. We also refer to the size/area of the region of a site as "the size/area of a site". The first algorithm we tested is shown in Algorithm 7. The aim of the algorithm is to provide a Voronoi site with enough space from the start, so that we can reduce the number of iterations in Algorithm 4. Unfortunately, Algorithm 7 is not able to do this reliably, because the Hilbert curve turns back on itself. Points later on the curve can be at a small distance from points early on the curve. This means that if we place a Voronoi site with a large desired area close to the center of the Hilbert curve square, it is very likely that another site will be placed close by. Figure 5.3 shows that even though the intervals between the sites on the Hilbert curve are equal, the distance between the sites is not. When we visualize the Euclidean distance between any two points on a Hilbert curve of order 8, using a heat map, we get the visualization as shown in Figure 5.4. This heat map can be seen as the matrix of distances with at the top-left corner the first point on the Hilbert curve. As we move to the right we move along the Hilbert curve until we reach the last point on the Hilbert curve at the top-right corner. The same happens moving down until we reach the last point on the Hilbert curve at the bottom-left corner. The brightness of the color shows the distance between two points; the brighter the color the larger the distance. We can clearly see that we might place sites closer together than intended.

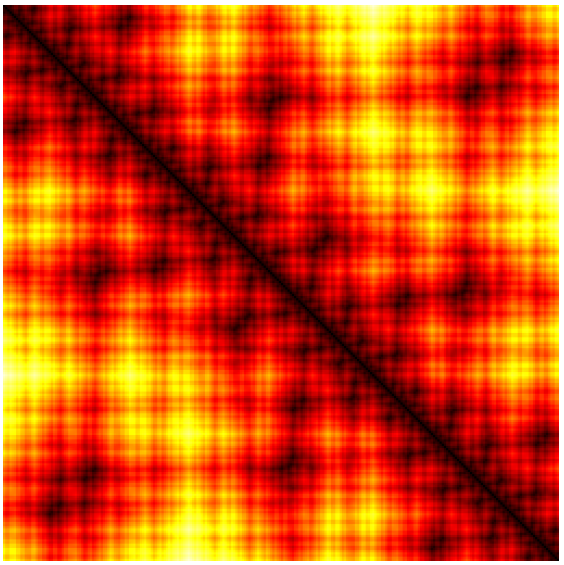

FIGURE 5.4: A heat map of the Euclidean distance between any two points on a Hilbert curve of order 8 (reproduced from datagenetics.com)

Data: A set $P = \{p_1, ..., p_i\}$, consisting of all points in the intersection of the Hilbert curve and some convex polygon
A hierarchical graph H with nodes with associated metric m
Result: A set $S = \{s_1, ..., s_j\}$ of Voronoi sites
begin
  $r \leftarrow$ get root node of H;
  $N \leftarrow$ child nodes of r in H;
  $S \leftarrow$ empty collection of Voronoi sites;
  $i \leftarrow$ number of points in P;
  $index \leftarrow 0$;
  foreach $n \in N$ **do**
    $fraction \leftarrow \frac{n_m}{r_m}$;
    $step \leftarrow fraction \times i$;
    Create Voronoi site s at $p_{index+\frac{step}{2}}$;
    $index \leftarrow index + step$;
    Add s to S;
  end
  return S;
end

Algorithm 7: Scaled Hilbert curve Voronoi sites

We solve this problem by using four Hilbert curves instead of a single Hilbert curve. When we create a new Voronoi site we determine which Hilbert curve to use based on the desired size of the region. The larger the desired area, the lower the order of Hilbert curve we use. This ensures that the location of new Voronoi sites will not accidentally lie close to other sites. In Figure 5.5, we can see that the lower the order of a Hilbert curve, the more guaranteed space each point has. We can use different order of Hilbert curves for different sites, because the general location of points in relation to each other

is preserved. In this new algorithm we do not first find all the points on the Hilbert curves that lie within a convex polygon, but we check each point we intend to use. Our algorithm uses the Hilbert curves of order 1, 2, 3, and 8. For each site, we determine which of the curves to use based on the percentage of the bounding square the desired area of a Voronoi site is. We then use that curve to find a suitable point that lies within the bounding polygon. If we cannot find one we fall back to a higher order curve. Once we reach the eighth order we keep trying till we find a suitable point. This gives us the algorithm as described in Algorithm 8. We should note that in Procedure FINDPOINTFORNODE we use a special Hilbert curve of order 1 (HC'_1): we added three points to the Hilbert curve to give us a little more choice when looking for a location for a new Voronoi site, see Figure 5.6.

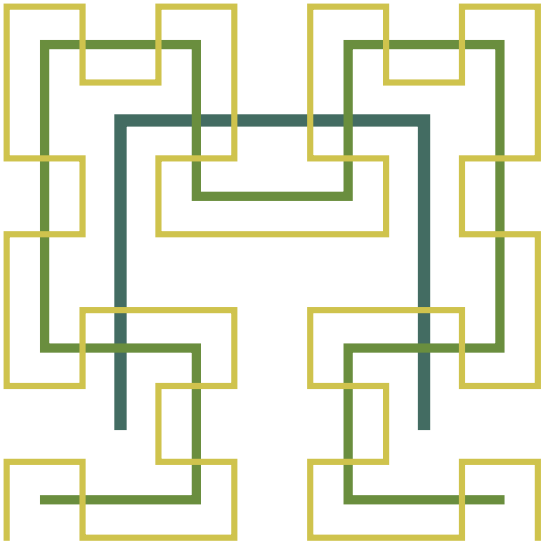

FIGURE 5.5: Three superimposed Hilbert curves of order 1, 2 and 3 (reproduced from wolfram.com)

5.4 Optimizing stable Voronoi treemap creation

Using the algorithms in Section 5.3, we found that we create good starting locations for the Voronoi sites when creating a single level of the Voronoi treemap. By also giving a good starting weight we could do even better. We originally started Algorithm 4 with sites that all have a weight of 1. We improve this by introducing an extra step in Algorithm 4 in which we compute a good starting weight. We do this by creating a traditional Voronoi diagram. Using this Voronoi diagram we check the area of the current Voronoi region of each site, and compare it with its desired area. We then change the weight based on the current area and how much the area needs to grow or shrink. When changing the weight we make sure that the circle of the site will not overlap

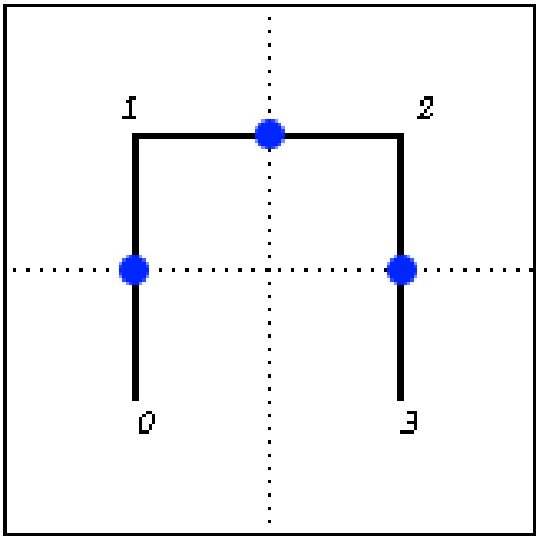

FIGURE 5.6: Three new points (blue) added to the Hilbert curve of order 1 (adapted from opengl.org)

with another site. Procedure COMPUTESTARTWEIGHTS shows how we compute the starting weight and Algorithm 9 shows how it is used.

Another optimization we did was improving the speed at which Voronoi sites with a small desired area reach their desired area. When a site reaches the lower bound of weight, it is totally dependent on its neighbors to increase their weight to get its desired area. We found that the second half of the iterations of Algorithm 9 was mostly spend on slowly propagating weight changes from all over the AWP Voronoi diagram to shrink the area of a few sites with a small desired area. We speed up this process by moving Voronoi sites with a small desired area that are at the lower bound of weight closer to one of its neighbors. By moving the site closer to its neighbor we ensure that the area shrinks, even if the weight does not change. The neighbor we move it closer to is the neighbor that has the smallest weight; this has the largest effect area wise. Procedure ADAPTPOSITIONSANDWEIGHTS' shows how we do this.

5.5 Incremental Voronoi treemap creation

We also tried to create stable Voronoi treemaps by using an incremental algorithm. To keep Voronoi treemaps as similar as possible, we create a new Voronoi treemap based on a previous one. When creating a Voronoi treemap we not only save the visualization, but also all the original data we used to create it, including the hierarchy of doubly-connected edge lists. When creating a new Voronoi treemap, we can provide the algorithms with a starting point. This starting point is the data and Voronoi treemap from an earlier visualization.

Data: The polygon P that has to contain the Voronoi sites
A hierarchical graph H with nodes with associated metric m
Result: A set $S = \{s_1, ..., s_j\}$ of Voronoi sites
begin
  $r \leftarrow$ get root node of H;
  $N \leftarrow$ child nodes of r in H;
  $S \leftarrow$ empty collection of Voronoi sites;
  $B \leftarrow$ the bounding perfect square of P;
  $areafraction \leftarrow \frac{P_{area}}{B_{area}}$;
  $indexfraction \leftarrow 0$;
  foreach $n \in N$ **do**
    $fraction \leftarrow \frac{n_m}{r_m} \times areafraction$;
    if $fraction <= 0.05$ **then**
      $(p, indexfraction) \leftarrow FINDPOINTFORNODE(P, r, n, fraction, 8)$;
    end
    if $fraction > 0.05$ *and* $fraction <= 0.10$ **then**
      $(p, indexfraction) \leftarrow FINDPOINTFORNODE(P, r, n, fraction, 3)$;
    end
    if $fraction > 0.10$ *and* $fraction <= 0.225$ **then**
      $(p, indexfraction) \leftarrow FINDPOINTFORNODE(P, r, n, fraction, 2)$;
    end
    if $fraction > 0.225$ **then**
      $(p, indexfraction) \leftarrow FINDPOINTFORNODE(P, r, n, fraction, 1)$;
    end
    Create Voronoi site s at p;
    Add s to S;
  end
  return S;
end

Algorithm 8: Variable Hilbert curve Voronoi sites

As Voronoi treemaps visualize hierarchical graphs, we can easily walk the graph of the new data and the graph of the previous visualization. While walking both graphs we only note the added and deleted nodes. We simply remove the deleted nodes from the previous Voronoi treemap. For nodes that are new, we look at the node that precedes them in the data, and look where it is in the visualization. We then introduce a new Voronoi site on the center of one of the edges of the region of the Voronoi site belonging to the preceding data, see Figure 5.7. If we can, we choose the edge that is shared with the Voronoi site that belongs to the data after the new node.

This approach does deliver very similar Voronoi treemaps when visualizing snapshots with only small differences. When old data is removed and new data is added in such a way that we cannot find edges to insert the new Voronoi sites, the algorithm breaks down. While we could have developed this algorithm further to better cope with these cases, we found that the Voronoi treemaps created by using variable Hilbert curves are

Data: P the polygon that has to contain the Voronoi sites
r root node with metric m
n node with metric m
$fraction$ the start fraction of the Hilbert curve
l the Hilbert curve order to start with
Result: A point p that lies in P
The current $fraction$
begin
  $HC'_1 \leftarrow$ the points on the extended Hilbert curve of order 1;
  $HC_2 \leftarrow$ the points on the Hilbert curve of order 2;
  $HC_3 \leftarrow$ the points on the Hilbert curve of order 3;
  $HC_8 \leftarrow$ the points on the Hilbert curve of order 8;
  $B \leftarrow$ the bounding perfect square of P;
  $stepfraction \leftarrow \frac{n_m}{r_m} \times \frac{P_{area}}{B_{area}}$;
  if $l == 8$ **then**
    $x \leftarrow SizeOf(HC_8)$;
    return $FINDPOINTFORNODEINCURVE(HC_8, P, r, n, fraction, x)$;
  end
  if $l == 3$ **then**
    $(p, fraction') \leftarrow$
     $FINDPOINTFORNODEINCURVE(HC_3, P, r, n, fraction, 8)$;
    if p *is null* **then**
      return $FINDPOINTFORNODE(P, r, n, fraction, 8)$;
    end
    return $(p, fraction')$;
  end
  if $l == 2$ **then**
    $(p, fraction') \leftarrow$
     $FINDPOINTFORNODEINCURVE(HC_2, P, r, n, fraction, 4)$;
    if p *is null* **then**
      return $FINDPOINTFORNODE(P, r, n, fraction, 3)$;
    end
    return $(p, fraction')$;
  end
  if $l == 1$ **then**
    $size \leftarrow SizeOf(HC'_1)$;
    $index \leftarrow size \times (fraction + \frac{stepfraction}{2})$;
    $p \leftarrow$ point in HC'_1 at $index$;
    if p *inside* P **then**
      return $(p, fraction + stepfraction)$;
    end
    $index \leftarrow index + 1$;
    $p \leftarrow$ point in HC'_1 at $index$;
    if p *inside* P **then**
      return $(p, fraction + stepfraction)$;
    end
    return $FINDPOINTFORNODE(P, r, n, fraction, 2)$;
  end
end

Procedure FINDPOINTFORNODE(P, r, n, $fraction$, l)

Data: HC a set of points on a Hilbert curve
P the polygon that has to contain the Voronoi sites
r root node with metric m
n node with metric m
$fraction$ the start fraction of the Hilbert curve
i_{max} max number of iterations
Result: A point p that lies in P or null
The current $fraction$
begin
  $B \leftarrow$ the bounding perfect square of P;
  $stepfraction \leftarrow \frac{n_m}{r_m} \times \frac{P_{area}}{B_{area}}$;
  $size \leftarrow SizeOf(HC)$;
  $index \leftarrow size \times (fraction + \frac{stepfraction}{2})$;
  $step \leftarrow size \times \frac{stepfraction}{2}$;
  $currentstep \leftarrow step$;
  $p \leftarrow$ point in HC at $index$;
  $i \leftarrow 0$;
  while p *not inside* P *and* $i < i_{max}$ **do**
    $i \leftarrow i + 1$;
    $currentstep \leftarrow \frac{currentstep}{2}$;
    if $currentstep < 1$ **then**
      $currentstep \leftarrow 1$;
    end
    $index \leftarrow index + currentstep$;
    $p \leftarrow$ point in HC at $index$;
  end
  if p *inside* P **then**
    return $(p, \frac{size}{index+step})$;
  end
  return $(null, 0)$;
end

Procedure FINDPOINTFORNODEINCURVE(HC, P, r, n, $fraction$, i_{max})

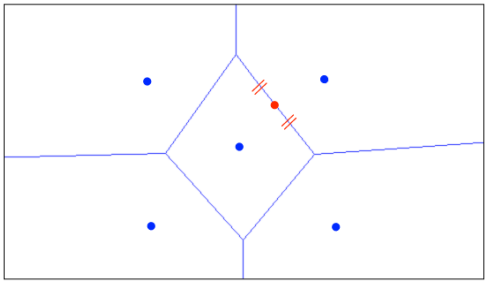

FIGURE 5.7: Inserting a new Voronoi site on the edge of the regions of two Voronoi sites

visually very close to what we could achieve by improving this algorithm. Another drawback of the incremental algorithm is that it needs a Voronoi treemap to start with, which means that we still have to use our variable Hilbert curve algorithm. Therefore

Data: A set $S = \{s_1, ..., s_n\}$ of weighted sites with no site in the circle of another site
Each site in S has an associated desired area
Each site in S has a starting weight of 1
A convex polygon P that bounds S
i_{max} max number of iterations
$E_{threshold}$ error threshold
Result: A doubly-connected edge list D bounded by P
begin
  $D \leftarrow$ initialize empty doubly-connected edge list;
  COMPUTESTARTWEIGHTS(S, P);
  for $i \leftarrow 1$ **to** i_{max} **do**
    $D \leftarrow$ create an AWP Voronoi diagram of S bounded by P;
    `// Please note we use the primed procedure`
    ADAPTPOSITIONSANDWEIGHTS'(D, S);
    $D \leftarrow$ create an AWP Voronoi diagram of S bounded by P;
    $totalError \leftarrow$ ADAPTWEIGHTS(D, S);
    $error \leftarrow \frac{totalError}{2 \times Area(P)}$;
    if $error < E_{threshold}$ **then**
      return D;
    end
  end
  return D;
end

Algorithm 9: Compute Voronoi treemap (single layer) improved

Data: A set $S = \{s_1, ..., s_n\}$ of weighted sites with no site in the circle of another site
P the bounding convex polygon
Result: All sites in S with updated weight
begin
  `// Please note we create a normal Voronoi diagram`
  $D \leftarrow$ create a Voronoi diagram of S bounded by P;
  $totalError \leftarrow 0$;
  foreach $s \in S$ **do**
    $nn \leftarrow$ nearest neighbor of s in D;
    $a_{current} \leftarrow$ current size of the Voronoi region of s in D;
    $a_{desired} \leftarrow$ desired size of the Voronoi region of s;
    $weight \leftarrow \frac{a_{current}}{\pi}$;
    $factor \leftarrow \frac{a_{desired}}{a_{current}}$;
    $newWeight \leftarrow min(weight \times factor, distance(s, nn)^2)$;
    Set weight of s to $max(newWeight, 1)$;
  end
  return $totalError$;
end

Procedure COMPUTESTARTWEIGHTS(S, P)

Data: A set $S = \{s_1, ..., s_n\}$ of weighted sites with no site in the circle of another site
Each site in S has an associated desired area
A bounded doubly-connected edge list D
Result: All sites in S updated with a new location and weight
begin
  foreach $s \in S$ **do**
    $weight \leftarrow$ weight of s;
    $a \leftarrow$ the Voronoi region of s in D;
    $cow \leftarrow$ the center of weight of a;
    Set location s at cow;
    if $weight == 1$ **then**
      $neighbor \leftarrow$ get neighbor of s with smallest weight in D;
      $borderDist \leftarrow$ get distance to edge between s and $neighbor$ in D;
      $a_{current} \leftarrow$ current size of the Voronoi region of s in D;
      $a_{desired} \leftarrow$ desired size of the Voronoi region of s;
      $factor \leftarrow \frac{a_{desired}}{a_{current}}$;
      $movedist \leftarrow borderDist \times factor$;
      Move s $movedist$ along the line segment between cow and $neighbor$;
    end
    else
      $minBorderDist \leftarrow$ min distance from s to edges in D;
      $newWeight \leftarrow min(\text{weight}, minBorderDist^2)$;
      Set weight of s to $newWeight$;
    end
  end
end

Procedure ADAPTPOSITIONSANDWEIGHTS'(D, S)

we decided to not further develop this incremental algorithm.

Chapter 6

Empirical validation

To be able to validate the visualizations obtained from our optimized stable Voronoi treemap algorithm, as described in Section 5.4, a web interface was build. The web interface allows a user to interactively create visualizations. To allow the users to get a complete overview of the visualizations, several example cases were provided. In this chapter an overview of the web interface and the example cases are given. The validation sessions with users will be described and the results discussed.

6.1 Web application

To provide the users with an easy way to interact with the algorithms and the resulting Voronoi treemaps, a web application was build. To create Voronoi treemaps, the algorithms need software analysis results; the application organizes these analysis results by software project and release, see Figure 6.1 and 6.2.

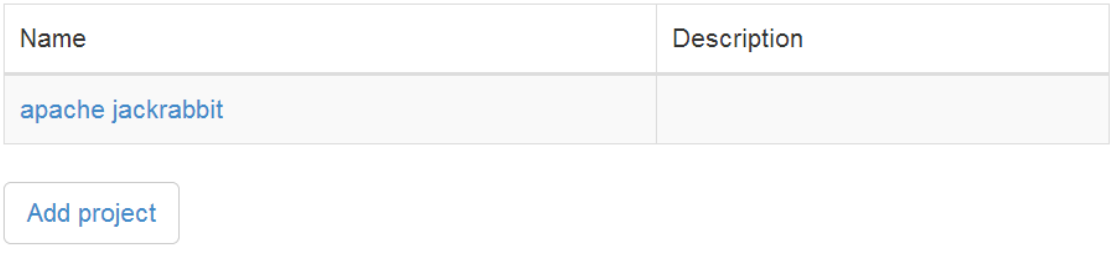

FIGURE 6.1: Web application project overview

6.1.1 Adding analysis results

To create visualizations, the algorithms need a hierarchical graph with metric values for each node in the graph. The application itself is not able to compute a graph and

FIGURE 6.2: Web application release overview

metrics. Therefore, they need to be supplied by an external tool. Since a web application is used, an upload function is provided to upload such a graph. The graph has to be in the format GEXF (Graph Exchange XML Format).

GEXF is an extensible XML-based file format that can describe complex network structures, their associated data and their dynamics. The format has a special construct to accommodate hierarchical network structures, as shown in Listing 6.1. As shown in Listing 6.2, attributes can be defined and added to the nodes. These two constructs are enough to provide the algorithms with enough data to create Voronoi treemaps.

The "label" attribute is used in the Voronoi treemaps as textual identification of a node. The "id" attribute is used to uniquely identify a node.

```
<?xml version="1.0" encoding="UTF-8"?>
<gexf xmlns="http://www.gexf.net/1.2draft"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.gexf.net/1.2draft http://www.gexf.net/1.2draft/gexf.xsd"
version="1.2">
     <graph mode="static" defaultedgetype="directed">
        <nodes>
          <node id="a" label="root">
            <nodes>
              <node id="b" label="child1">
                <nodes>
                  <node id="c" label="grandchild1"/>
                  <node id="d" label="grandchild2"/>
                </nodes>
              </node>
              <node id="e" label="child2">
                <nodes>
                  <node id="f" label="grandchild3"/>
                </nodes>
              </node>
            </nodes>
          </node>
        </nodes>
    </graph>
</gexf>
```

Listing 6.1: Hierarchical GEXF

```
<?xml version="1.0" encoding="UTF-8"?>
<gexf xmlns="http://www.gexf.net/1.2draft"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.gexf.net/1.2draft http://www.gexf.net/1.2draft/gexf.xsd"
version="1.2">
    <meta lastmodifieddate="2009-03-20">
        <creator>Gephi.org</creator>
        <description>A Web network</description>
    </meta>
    <graph defaultedgetype="directed">
        <attributes class="node">
            <attribute id="0" title="url" type="string"/>
            <attribute id="1" title="indegree" type="float"/>
            <attribute id="2" title="type" type="int">
                <default>true</default>
            </attribute>
        </attributes>
        <nodes>
            <node id="3" label="BarabasiLab">
                <attvalues>
                    <attvalue for="0" value="http://barabasilab.com"/>
                    <attvalue for="1" value="3.14"/>
                    <attvalue for="2" value="1"/>
                </attvalues>
            </node>
        </nodes>
    </graph>
</gexf>
```

Listing 6.2: Data GEXF

To actually upload a GEXF file, a .gexf file has to be selected and a description can be provided, as shown in Figure 6.3. The file will be shown in the release overview, where it can be used to create Voronoi treemaps.

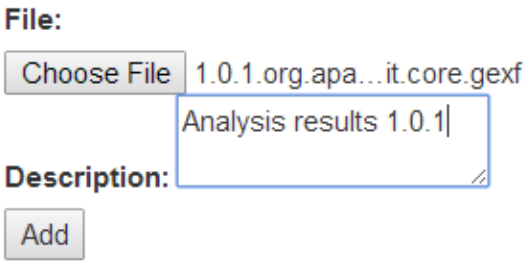

Figure 6.3: Web application adding analysis results

6.1.2 Creating a Voronoi treemap

Voronoi treemaps are created by selecting the "Add visualization" link for a particular analysis result, as shown in Figure 6.4. The user is then presented with a screen where

the size and sorting attributes can be selected, as seen in Figure 6.5 and 6.6. All the attributes that are available for selection are the metrics that have been provided in the uploaded GEXF file. The size attribute will be used to compute the relative size of the areas for each node, as described in Chapter 4. To create a stable Voronoi treemap the sorting attribute is used as described in Chapter 5. By clicking the "Add" button, the selected options and the graph will be fed to the algorithms and the resulting Voronoi diagram will be shown.

FIGURE 6.4: Web application analysis results overview

FIGURE 6.5: Available size attributes in web application "Add visualization"

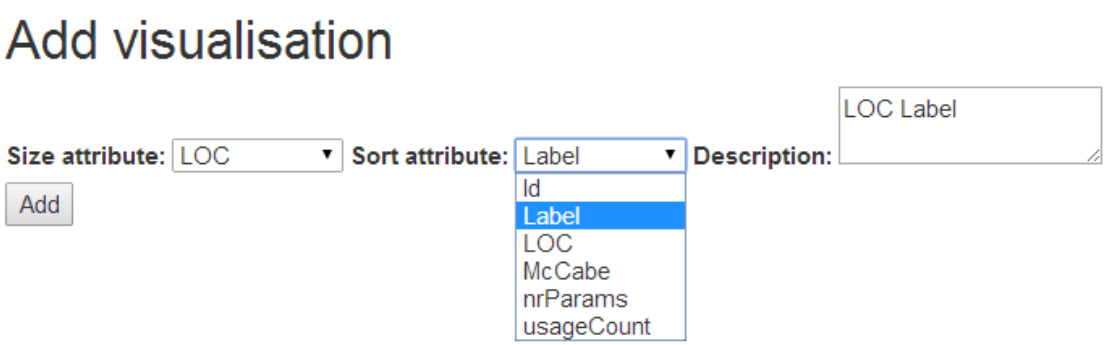

FIGURE 6.6: Available sort attributes in web application "Add visualization"

6.1.3 Interacting with a Voronoi treemap

Once a Voronoi treemap has been created, it will be listed in the overview beneath the corresponding GEXF file. By selecting it, the user is taken to a full-screen version of the

Voronoi treemap. The treemap is rendered as a Scalable Vector Graphics (SVG) picture. The use of a vector graphics means that zooming in will not result in artifacts and loss of detail. Figure 6.7 shows a part of the same Voronoi treemap at different zoom levels. When hovering over an area in the Voronoi treemap the corresponding label is shown, as seen in Figure 6.8. Clicking on an area will show all the information we have on the corresponding node; all the attributes are shown with their values. Because it is only possible to click on leaf nodes, we also provide a tree list of the ancestors of the selected node.

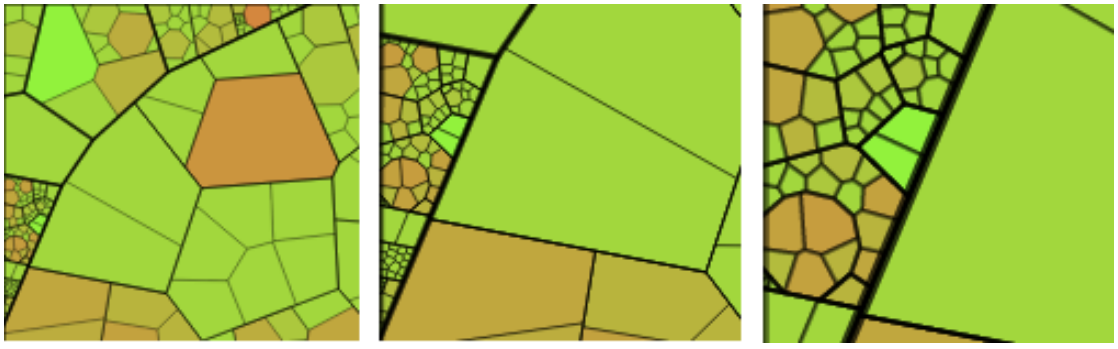

FIGURE 6.7: Voronoi treemap at several zoom levels

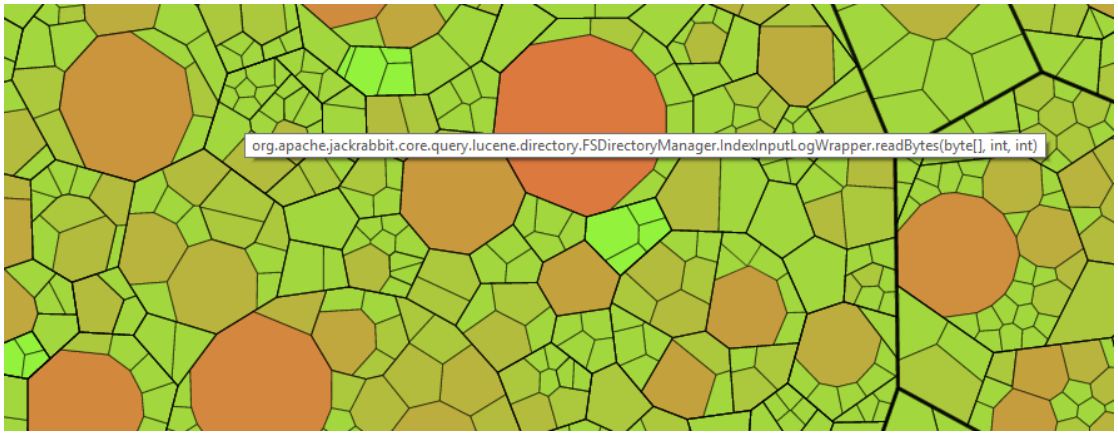

FIGURE 6.8: A tooltip shown when hovering over a Voronoi treemap

By default, the colors of the Voronoi treemap represent the McCabe complexity of each leaf node. This can be changed by selecting another attribute in the color attribute drop-down. By giving the minimal and maximal value expected for the selected attribute, a range is computed from green to red. The minimal value will be green and the maximal red.

6.1.4 Comparing Voronoi treemaps

Comparing two Voronoi treemaps is done by selecting one and clicking the "Compare" button. This brings up a screen to select a Voronoi treemap to compare with. On

this screen the user can also select the attribute to use for the comparison. By clicking "OK" the originally selected Voronoi treemap is shown, but the color now represents the difference between the two Voronoi treemaps for the selected attribute. The intensity in blue shows decrease in percentage. The intensity in orange shows the increase in percentage. White means no change and green means that the node was not present in the Voronoi treemap that is selected to compare with. Figure 6.9 and 6.10 show two examples.

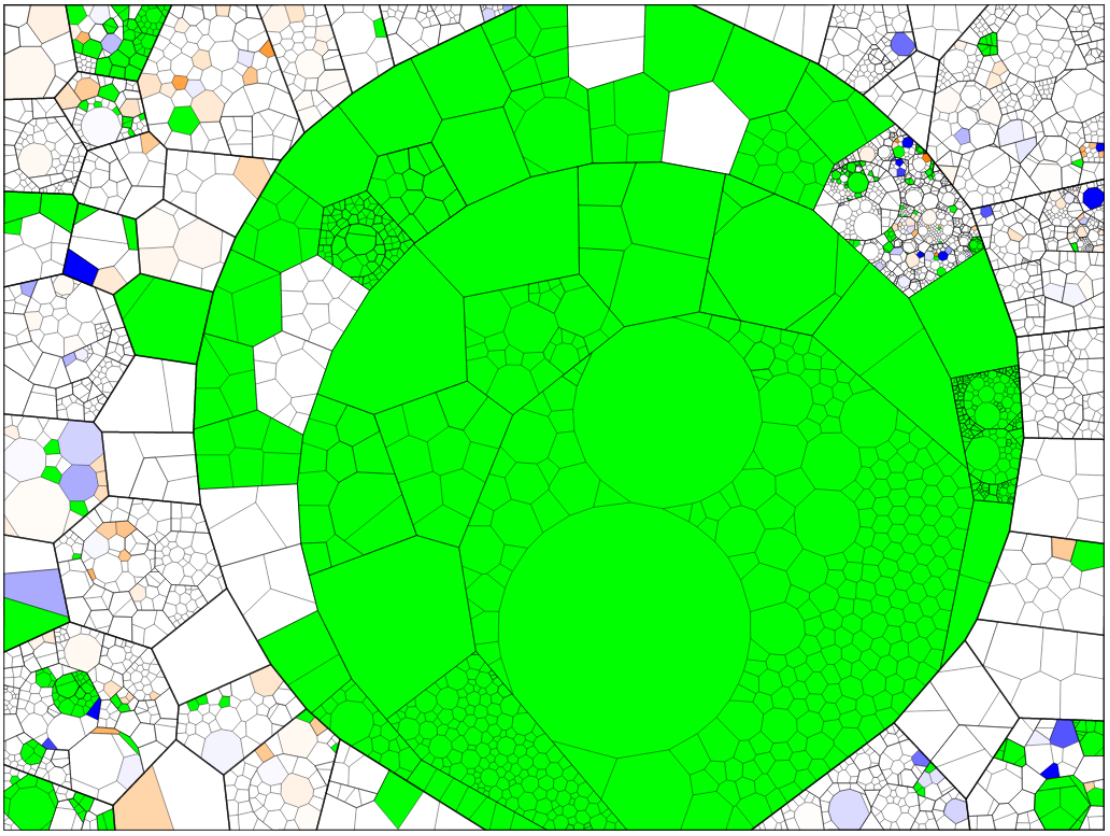

FIGURE 6.9: The difference between Apache Jackrabbit 1.5.0 and 1.6.0

The available interaction is the same as with a normal Voronoi treemap, except when selecting a node, not only the current values are shown, but also the delta.

6.2 Validation cases

Because of the short time limit set for the empirical validation two example cases were selected, instead of using systems that the participants had prior experience with. We used the data set [33], which contains analysis results from the Maven repository. For each project in the Maven repository are analysis results for several versions available. Two representative projects were selected and extensively studied, so that they could be used in the empirical validation: Checkstyle and Apache Jackrabbit Core.

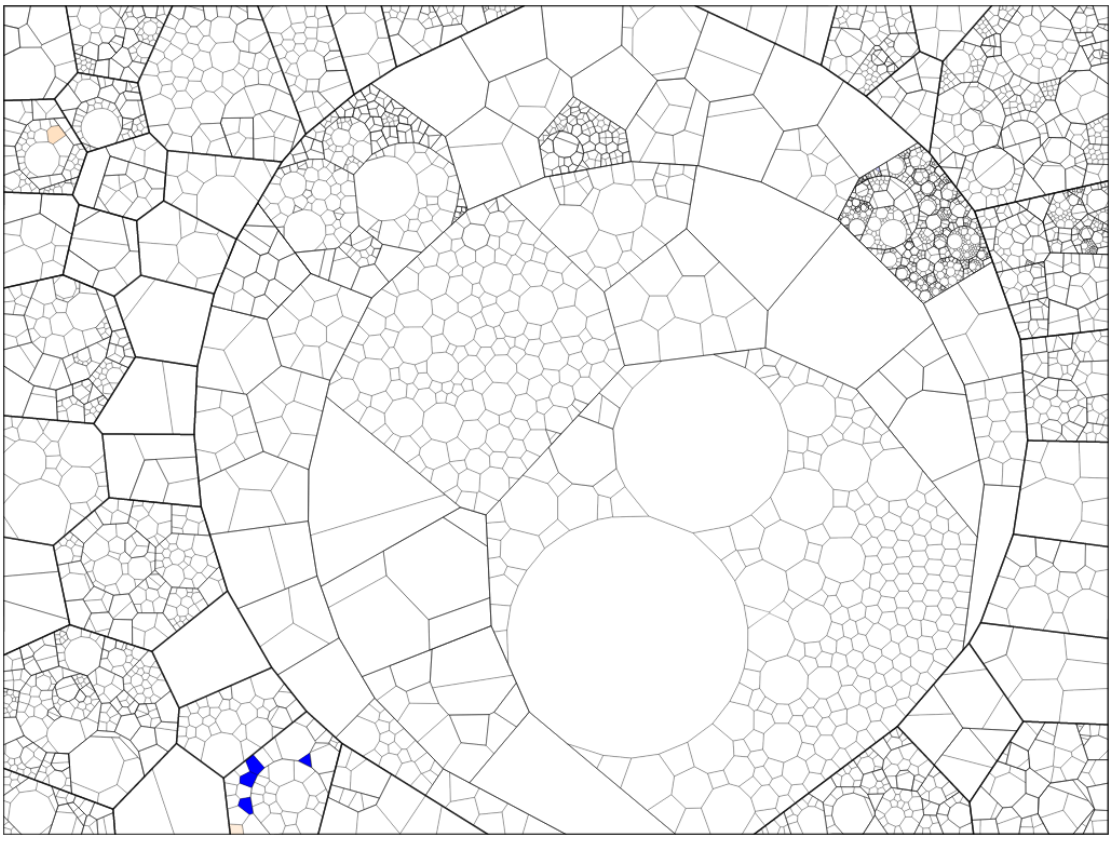

FIGURE 6.10: The difference between Apache Jackrabbit 2.0.0-beta1 and 2.0.0-beta3

6.2.1 Checkstyle

Checkstyle is a development tool to help programmers write Java code that adheres to a coding standard. It automates the process of checking Java code to spare humans of this boring (but important) task. Historically, its main functionality has been to check code layout issues, but since the internal architecture was changed in version 3, more and more checks for other purposes have been added. Now, Checkstyle provides checks that find class design problems, duplicate code, or bug patterns like double checked locking. Checkstyle provides a beautiful code base that has two interesting features that makes it ideal for validating the visualizations.

First, Checkstyle has both generated code and handwritten code. The two are easily distinguished in the visualizations. In Figure 6.11, Checkstyle version 5.4 is visualized with lines of code as size and McCabe complexity as color. Generated code shows as large bright red regions and handwritten code as small green regions.

Second, with around 18.000 lines of beautiful handwritten code, Checkstyle can easily be used to check conclusions drawn from the visualizations. As expected, the generated code is not as beautiful, but does serve as a good example of complex code.

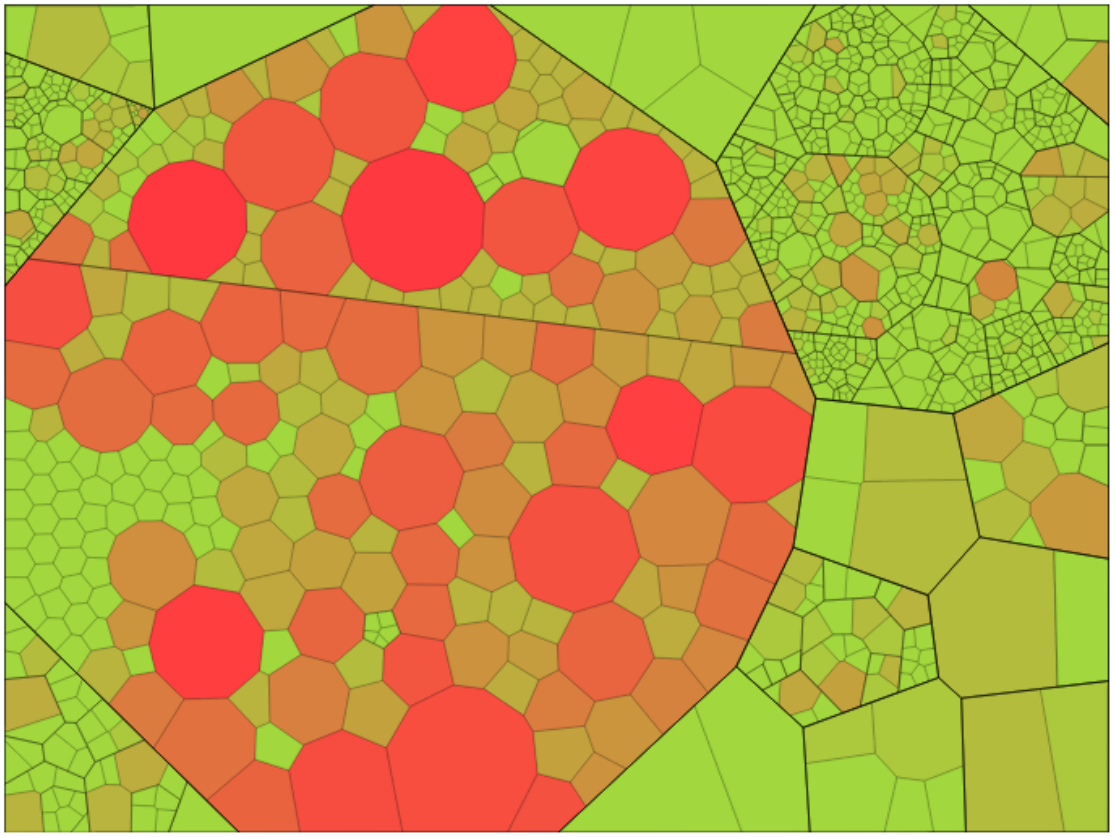

FIGURE 6.11: Checkstyle 5.4

6.2.2 Apache Jackrabbit Core

Apache Jackrabbit Core is the core component of the Apache Jackrabbit project. Apache Jackrabbit is a fully featured content repository that implements the entire JCR API. With 265.000 lines of code, Apache Jackrabbit Core is a large software system, and with over 35 snapshots in the SIG data set, a good fit for our empirical validation.

The snapshots range from version 1.0.0 to 2.2.7 and show a good amount of evolution between the versions. The snapshots also contain a run of versions that are close together: 2.0.0-beta1 through 2.0.0 final with four betas in between.

The stability of the Voronoi treemaps can easily be shown by visualizing these snapshots. Figure 6.12, 6.13 and 6.14 show three beta versions of Apache Jackrabbit 2.0.0. Figure 6.15 shows the final 2.0.0 version. There are only small changes between beta1 and beta3; the visualizations reflect this by being very similar. The visualizations of beta3 and beta5 show a bigger difference, which is due to the fact that there is a bigger difference in the software. Even with this bigger difference the two visualizations are very similar.

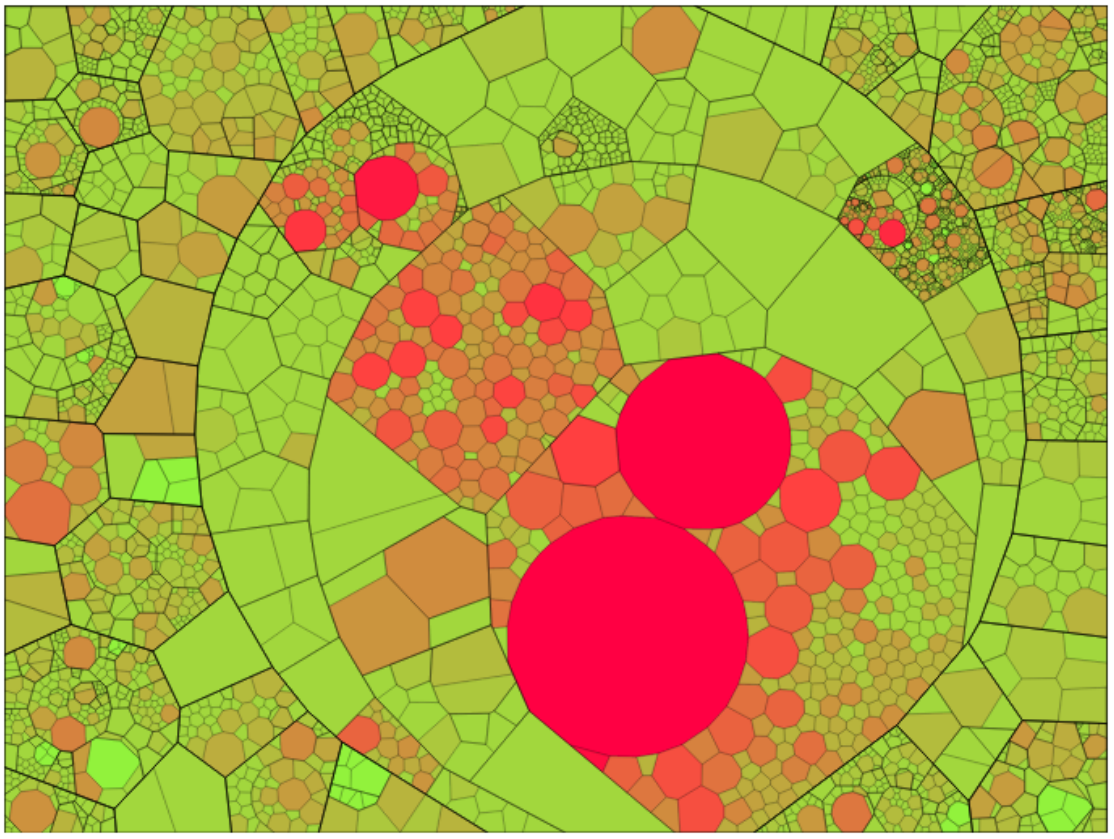

FIGURE 6.12: Apache Jackrabbit 2.0.0-beta1

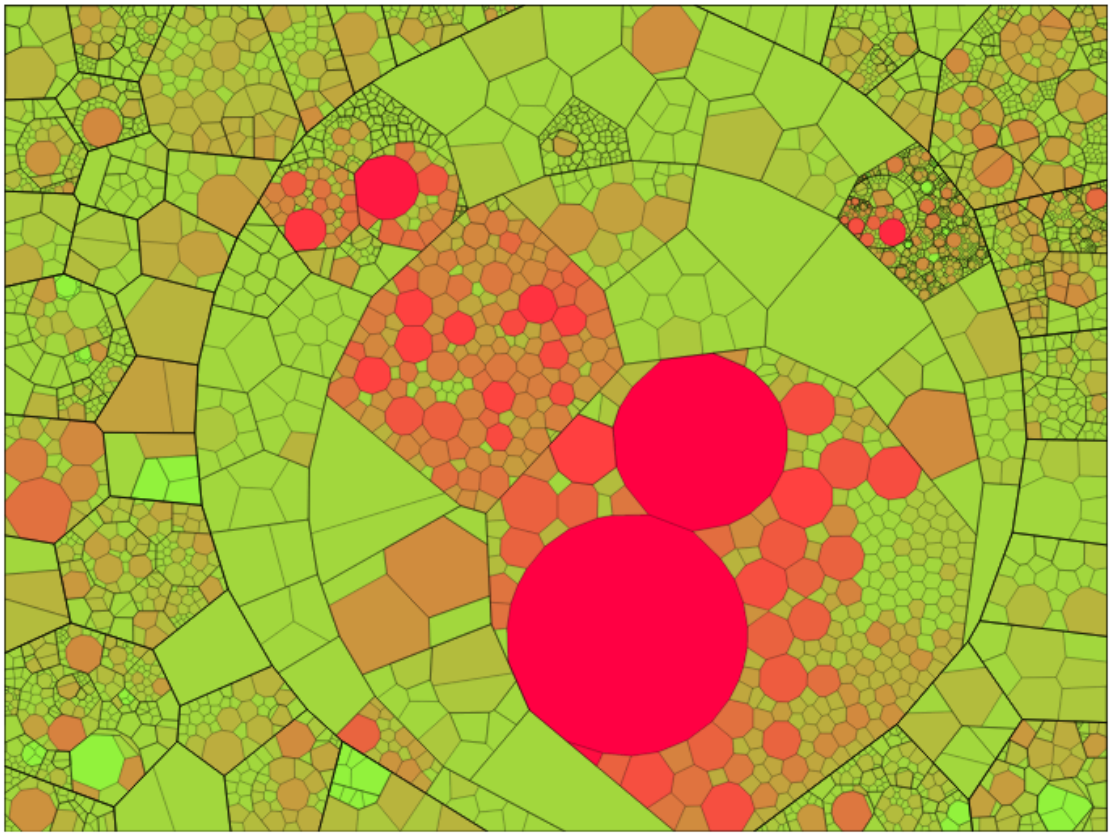

FIGURE 6.13: Apache Jackrabbit 2.0.0-beta3

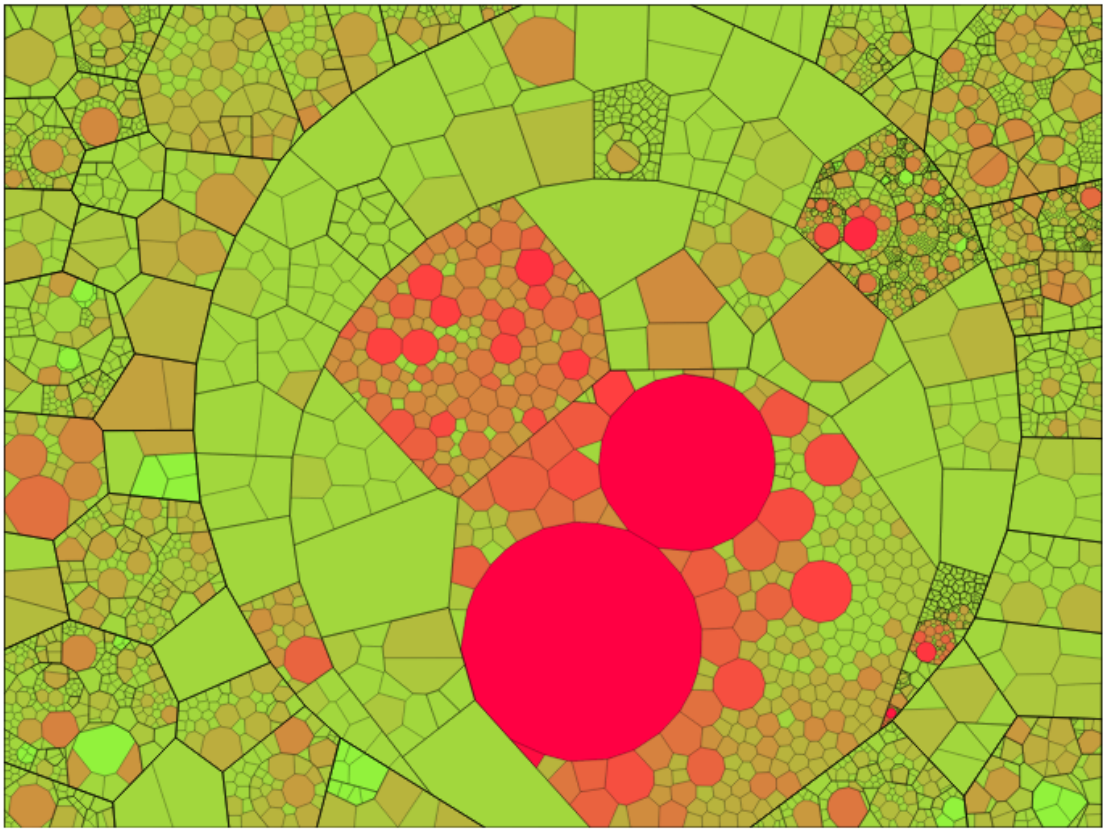

FIGURE 6.14: Apache Jackrabbit 2.0.0-beta5

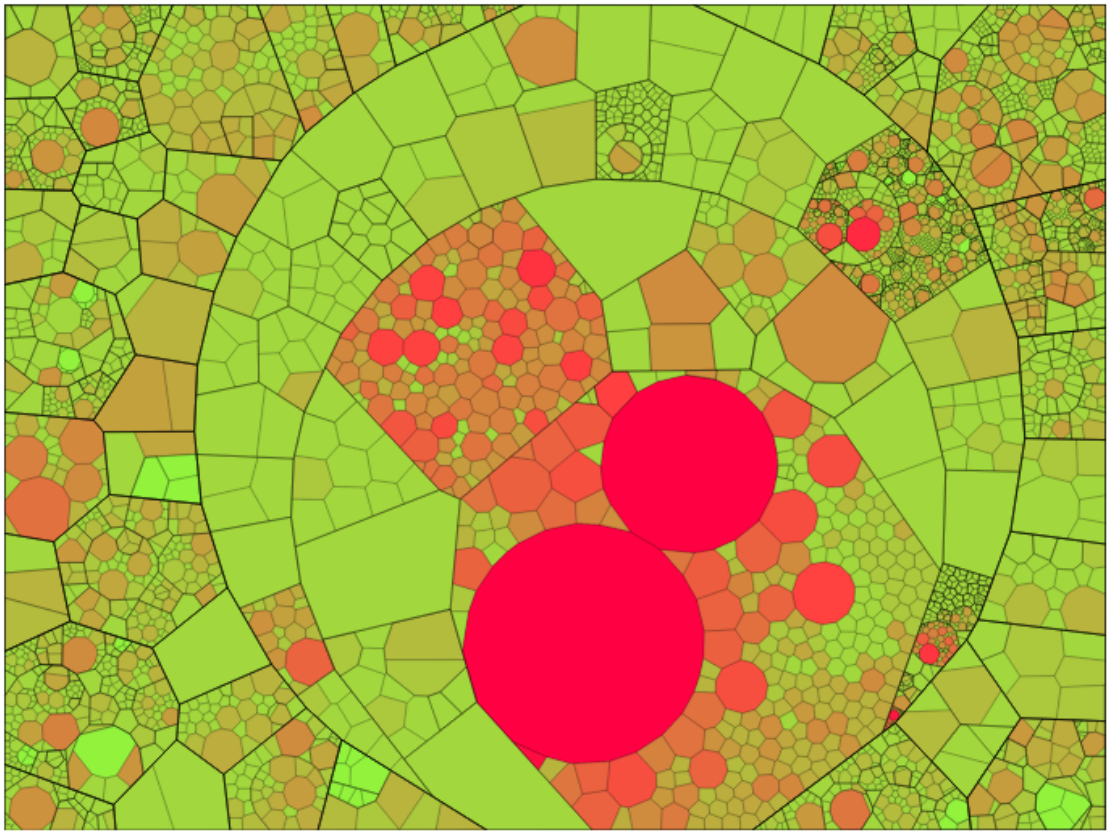

FIGURE 6.15: Apache Jackrabbit 2.0.0

6.3 User sessions

Because Voronoi treemaps might be valuable at several stages in software development and software quality assessment, people with different backgrounds were selected to participate in the empirical validation. The backgrounds ranged from project manager to software quality assessor. Validation was done in two different types of sessions: interactive sessions and demo sessions. In Table 6.1 we give an overview of the sessions.

During each session, three main areas of interest were discussed: the value of Voronoi treemaps themselves, the added value of stability when visualizing different versions of the same software system, and the added benefit of speed.

Session type	No. persons	Duration (min)	Technology	Experience (years)	Team role
Demo	1	45	.Net	> 12.5	Software architect
Interactive	1	75	.Net	> 12.5	Software architect
Demo	1	60	.Net	> 12.5	Project lead
Demo	2	60	.Net Java	> 12.5 > 12.5	Project lead Software architect
Interactive	1	75	.Net	> 12.5	Software architect
Interactive	1	45	.Net	> 5	Lead developer, scrum master
Demo	2	45	– .Net	> 12.5 > 7.5	Project manager Technical analyst
Demo	1	60	.Net, Java	> 25	Technical analyst, software quality assessor

TABLE 6.1: Overview of user sessions

6.3.1 Voronoi treemap validation

While publications about Voronoi treemaps have been around for some time, an empirical validation of the visualizations has never been published. Therefore, it was decided that it would be beneficial to include a validation of Voronoi treemaps as starting point for the validation of the rest of this master thesis project.

At the beginning of each validation session, the participant was given a short introduction to Voronoi treemaps, including how they visualize hierarchical graphs and use size to depict one metric and color as a second metric. After this short introduction, several example Voronoi treemaps were shown. These example Voronoi treemaps came from both Checkstyle and Apache Jackrabbit Core.

For each Voronoi treemap, the value of the visualization itself was discussed.

6.3.2 Speed validation

One of the benefits of the algorithms described in this thesis is the speed. Creating a Voronoi treemap for analysis results with a specific set of metrics takes mere seconds. During the interactive sessions participants were asked to form an opinion on one of the example cases. They were given the analysis results, access to the source code of the system and the web application, and asked to explore the system and form an opinion of the system.

During 30 minutes the interactions of the participant and the web application were observed. After 30 minutes the results were discussed.

For non-interactive sessions the participants were given the option to create a few Voronoi treemaps, but not without previously been given an explanation of the system being visualized.

6.3.3 Stability validation

The most important addition to Voronoi treemaps that this thesis proposes is the addition of stability. To verify if our algorithm provides useful stability, the participants of validation sessions were shown multiple Voronoi treemaps of different versions of a system. They were also shown comparisons between Voronoi treemaps. The benefits of the stability were discussed after participants had a chance to create and look at multiple Voronoi treemaps.

6.3.4 User validation strengths and limitations

We tried to get a fair and balanced validation of our results, but we identified several limitations in our validation setup. We also identified several strengths. Taking these strengths and limitations into account we consider the results of our empirical study valid.

Limitations:

- All validation sessions were done with people from the same company. Even though the company in question prides itself in its rigorous quality guidelines, it does mean that all participants view our results within those guidelines and company culture.
- Most of the user sessions were conducted with people with a .Net background while our example cases were in Java. Java and .Net are very similar and the participants could still follow the code, but unfortunately the finer details might have been lost to them.
- None of the participants had prior knowledge of the example software systems we used. By studying the software ourselves we were able to answer most questions the participants had.

Strengths:

- All participants in the user sessions have at least 5 years of experience with software development; most have more than 12.5 years of experience.
- The user sessions were conducted with people that have different backgrounds and roles. We tried to get a cross section of all people that have a direct stake in monitoring software quality.
- Several participants have done software quality assessments or were directly involved in software quality control.
- We chose representative example software systems from the available systems in [33].
- The two example systems show a wide range of situations, from beautiful hand-written code to extremely complex generated code.

6.4 Validation results

In each validation session valuable remarks were made. The most important and recurring remarks are summarized here.

- Voronoi treemaps give a good bird's-eye view of a software system.
- A single Voronoi treemap is not enough to show software quality, but offers a good starting point for investigation or discussion.

- The possibility to create many Voronoi treemaps showing different metrics of a system in a short time is a strong feature when investigating a new software system. Especially when taking over development or maintenance of an existing software system.
- The speed and stability of the algorithms help to quickly apply the gained insight of one visualization to another visualization.
- The stabilized Voronoi treemaps allow for continuous feedback on the development of software systems. This is especially valuable in larger teams on products that are developed over a longer period of time. Accumulating disjointed small changes can inadvertently lead to degrading software quality; by creating Voronoi treemaps at small intervals this can be identified early on.
- The ability to compare different versions of the same software system is valuable when keeping track of the development process and the quality of the resulting software.
- Comparing two Voronoi treemaps with respect to absolute sizes is hard. The sizes of areas in a Voronoi treemap are relative to the other areas of the same Voronoi treemap. But not relative to the areas in a different Voronoi treemap. To solve this, the absolute size of the system could be used to scale the complete Voronoi treemap. This would allow for correctly comparing sizes of areas between two Voronoi treemaps.

Given these results we can conclude that stable Voronoi treemaps fulfill all the requirements presented in Section 1.1. The participants in validation sessions also noted that there was room for improvement. These improvements were not in the algorithms or Voronoi treemaps themselves, but in the web application used to interact with them, or in the metrics available for the visualizations.

Chapter 7

Conclusion and future work

In this master thesis we introduced interactive stable Voronoi treemaps. To be able to create these interactive stable Voronoi treemaps we developed two novel algorithms: a sweep line algorithm for additively weighted power Voronoi diagrams and an algorithm that creates stable Voronoi treemaps. We verified the usefulness of the interactive stable Voronoi treemaps by doing an empirical study.

7.1 Sweep line algorithm for additively weighted power Voronoi diagrams

One of the contributions of this thesis is a sweep line algorithm for additively weighted power Voronoi diagrams. Using this algorithm to create Voronoi treemaps and verifying that these treemaps are correct, we indirectly showed that the AWP Voronoi diagrams we created are correct. The empirical study also showed that our algorithm is fast enough to provide an interactive environment for stable Voronoi treemaps.

We never compared our algorithm to other known algorithms for AWP Voronoi diagrams. We think that it would be useful to do these comparisons, especially on run time and complexity. We also suspect that our algorithm can be simplified by further examining the top site circle event and its interaction with both the sweep line and the beach line.

7.2 Stable Voronoi treemap algorithm

The main contribution of this thesis is the introduction of stability to Voronoi treemaps. We created several algorithms that are able to provide stability and finally settled on

the algorithm based on scaled Hilbert curves. By using a web application that enables users to create and interact with stable Voronoi treemaps, we were able to show that our implementation of stable Voronoi treemaps meet the requirements set in Section 1.1. Stable Voronoi treemaps are able to convey software quality and are especially useful in software quality monitoring.

We also showed that using our scaled Hilbert curve algorithm we are able to improve the time it takes to create Voronoi treemaps. This improvement was achieved by introducing an algorithm that creates a better starting weight for the Voronoi sites. This in combination with the sweep line AWP Voronoi diagram algorithm allows us to create stable Voronoi treemaps fast enough for users to quickly explore the quality of a software system. This was confirmed in the empirical validation.

The scaled Hilbert curve algorithm produces stable Voronoi diagrams, but was never analyzed in depth. We think that the constants currently present in the algorithm could be improved on. A study into the relation between these constants and the stability of the resulting treemap could further improve the stability of Voronoi treemaps. We also foresee that the order of Hilbert curves we use can be improved upon, not only to improve the stability but also to create a better starting situation for the other algorithms.

Appendix A

Implementation details

The implementation of our application used in Chapter 6 can be divided in two separate parts: the part that creates a stable Voronoi treemap and the part that provides all the needed data and displays the result. Creating a stable Voronoi treemap is done by using the library described in Section A.1. The web application described in Section A.2 is used to interact with the library.

A.1 Stable Voronoi treemap library

The stable Voronoi treemap library is written in Java and has no dependencies on other libraries. The library exposes several public functions, the most important of which is the creation of stable Voronoi treemaps. It also exposes functions to create bounded and unbounded AWP Voronoi diagrams. To create a stable Voronoi treemap we have to provide the library with a hierarchical set of Voronoi sites and a bounding convex polygon. The Voronoi sites do not need a location or a weight, only a desired size. The location and the weight will be computed by the library. The Voronoi sites also have a reference to the data that they represent; this data can later be used in the visualizations.

The library has a total size of 3760 source lines of code (SLOC). All the functionality that is exposed is implemented as static functions. This means that the functions are thread-safe and can be used concurrently. Figure A.1 shows the structure of the library. For each file we show the source lines of code and give a short description of its functionality when it is not directly apparent from its name.

```
src/main/java
├── AWPBoundedVoronoi.java ..... 251 SLOC
│       Creates an AWP Voronoi diagram bounded by a convex polygon
│       for a set of weighted Voronoi sites.
├── AWPTreeMapVoronoi.java ..... 270 SLOC
│       Creates a stable Voronoi treemap bounded by a convex
│       polygon for a hierarchical data set of weighted Voronoi
│       sites with associated desired area.  Implements Algorithm
│       9.
├── AWPVoronoi.java ..... 95 SLOC
│       Creates a Voronoi diagram for a set of weighted Voronoi
│       sites.  Implements Algorithm 2.
├── BeachArc.java ..... 284 SLOC
│       Represents a beach arc in the beach line.  Implements the
│       logic for finding beach arc intersections as described in
│       Subsection 3.3.2.3 and Subsection 3.3.2.5.
├── BeachLine.java ..... 306 SLOC
│       Represents the beach line.  Implements the logic for
│       finding a beach arc directly above a site and the logic
│       for inserting a new arc into the beach line as described
│       in Subsection 3.3.2.3.
├── Circle.java ..... 114 SLOC
│       Used as circle event.
├── ConvexPolygon.java ..... 292 SLOC
├── Edge.java ..... 115 SLOC
├── HalfEdge.java ..... 64 SLOC
├── Circle.java ..... 378 SLOC
│       Used to place Voronoi sites using multiple order of
│       Hilbert curves.  Implements Algorithm 8.
├── Point.java ..... 56 SLOC
├── Site.java ..... 153 SLOC
│       A weighted Voronoi site.  Also used as site event and as
│       TSC event.
├── SweepArc.java ..... 51 SLOC
│       Part of the sweep line that holds a reference to a Voronoi
│       site that influences the sweep line.
└── SweepLine.java ..... 128 SLOC
        Holds the current y-coordinate of the sweep line and a
        collection of SweepArcs.  Is used to determine if a circle
        is above or below the sweep line.
```

FIGURE A.1: Stable Voronoi treemap module

```
├── TreeSite.java ..... 94 SLOC
│       Extends the normal Voronoi site to support a hierarchical
│       structure.
├── VoronoiUtil.java ..... 669 SLOC
│       Utility class that holds all the mathematical functions
│       that are needed to compute Voronoi treemaps.
└── redblacktree
    ├── RedBlackTree.java ..... 334 SLOC
    │       An implementation of red-black trees that exposes parts
    │       of its internals so we can use it in our implementation of
    │       the beach line.
    └── RedBlackTreeNode.java ..... 106 SLOC
            The nodes for the red-black tree, used by the beach arcs.
```

FIGURE A.1: Stable Voronoi treemap module continued

A.2 Web application

The web application that we developed to interact with the stable Voronoi treemap library allows users to create and interact with stable Voronoi treemaps using their browser. The functionality of the application is explained in Chapter 6. The application architecture is based on a RESTful Model-View-Controller (MVC) pattern and can be divided in a front-end and a back-end. The front-end is the HTML and JavaScript used by the browser. The back-end provides the data and interacts with the stable Voronoi treemap library.

The web application uses many standard components; Table A.1 shows each component and how it is used. In Figure A.2 the structure of the back-end code is shown. For each directory we show the total source lines of code and the function of the code in the directory. The back-end is completely written in Java and has a total of 2141 source lines of code written by us. The front-end structure is shown in Figure A.3. Again we show the function of the contents of each directory, but we only show the source lines of code of JavaScript that we have written. In total, we have written 295 source lines of JavaScript code.

Project	Function	URL
Spring Boot	Java MVC framework with embedded web server. Forms the basis of our back-end.	projects.spring.io/spring-boot/
Bootstrap 3.0	HTML, CSS and JS framework. Used to create our HTML interface.	getbootstrap.com
Thymeleaf	A modern Java HTML templating engine. Used to create the HTML send to the browser.	thymeleaf.org
Gephi Toolkit	Stand-alone toolkit that exposes Gephi features as a Java library. Used to read .GEXF files.	wiki.gephi.org/index.php/Toolkit_portal
D3js	JavaScript library for manipulating documents based on data. Used to render the stable Voronoi treemaps and provide interactivity.	d3js.org
gexf-parser	JavaScript library that reads .GEXF files and exposes them as plain JS objects. Used to interact with analysis result data.	github.com/Yomguithereal/gexf-parser
underscore.js	JavaScript library that provides functional programming helpers. Used to efficiently handle the object graphs.	underscorejs.org

TABLE A.1: Libraries and frameworks used in the web application

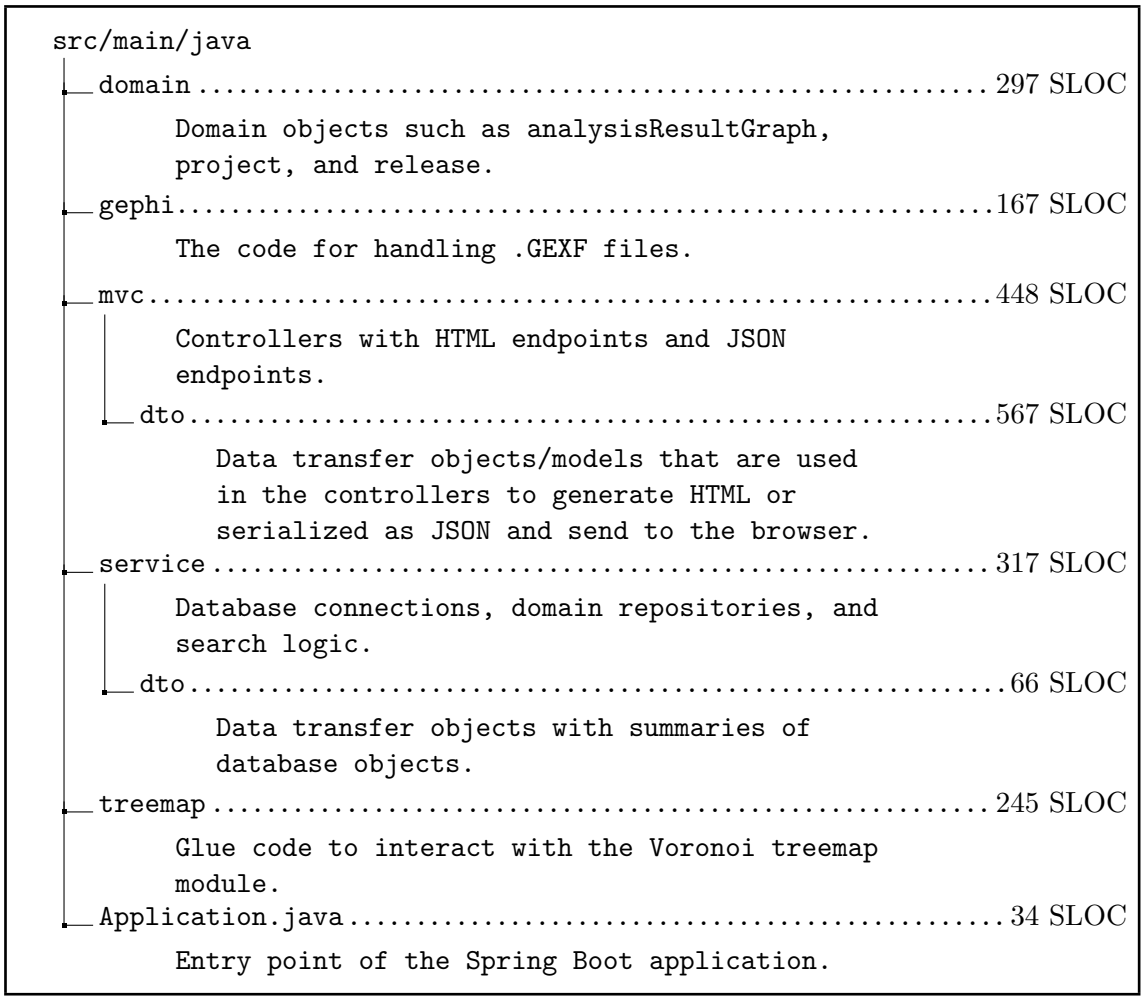

FIGURE A.2: Web application back-end

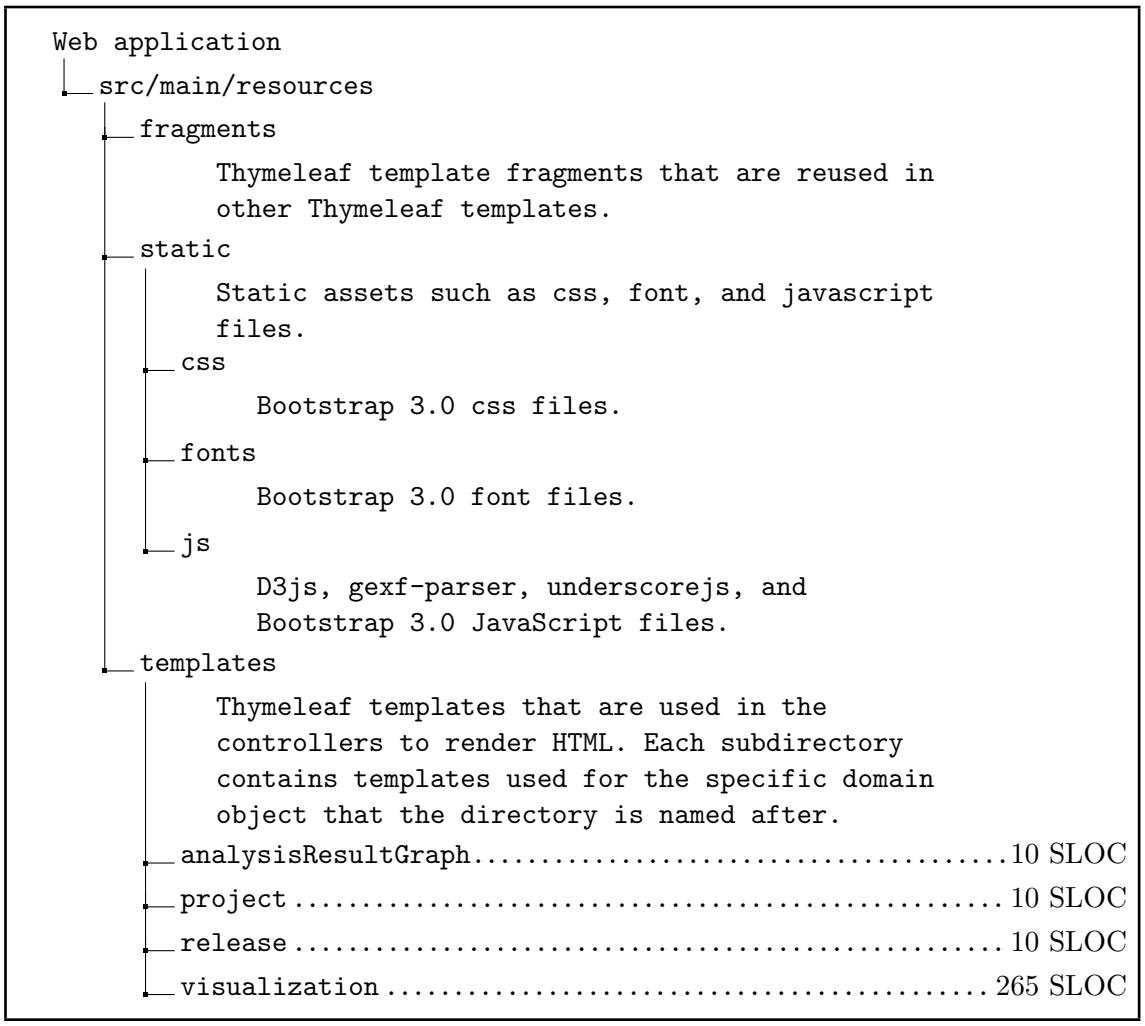

FIGURE A.3: Web application front-end

Bibliography

[1] JJ van Wijk. Views on Visualization. *Visualization and Computer Graphics, IEEE Transactions on*, 12(4):1000–433, 2006.

[2] Michael Balzer, Oliver Deussen, and Claus Lewerentz. Voronoi treemaps for the visualization of software metrics. In *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, pages 165–172, New York, NY, USA, 2005. ACM Press. ISBN 1-59593-073-6. doi: http://doi.acm.org/10.1145/1056018.1056041.

[3] B. Cornelissen, A. van Deursen, L. Moonen, and A. Zaidman. Visualizing Testsuites to Aid in Software Understanding.

[4] P. Gestwicki and B. Jayaraman. Methodology and architecture of JIVE. *Proceedings of the 2005 ACM symposium on Software visualization*, pages 95–104, 2005.

[5] David Gleich, Metthew Rasmussen, Kevin Lang, and Leonid Zhukov. The world of music: User ratings; spectral and spherical embeddings; map projections.

[6] D. Holten. Hierarchical Edge Bundles: Visualization of Adjacency Relations in Hierarchical Data. *Proceedings of the IEEE Symposium on Information Visualization. IEEE Computer Society*, 2006.

[7] A. Noack and C. Lewerentz. A space of layout styles for hierarchical graph models of software systems. *Proceedings of the 2005 ACM symposium on Software visualization*, pages 155–164, 2005.

[8] M. Pinzger, H. Gall, M. Fischer, and M. Lanza. Visualizing multiple evolution metrics. *Proceedings of the 2005 ACM symposium on Software visualization*, pages 67–75, 2005.

[9] A. Telea, A. Maccari, and C. Riva. An Open Visualization Toolkit for Reverse Architecting. *Proceedings of the 10th International Workshop on Program Comprehension*, 2002.

[10] Q. Wang, W. Wang, R. Brown, K. Driesen, B. Dufour, L. Hendren, and C. Verbrugge. EVolve: an open extensible software visualization framework. *Proceedings of the 2003 ACM symposium on Software visualization*, 2003.

[11] M. Bruls, K. Huizing, and J.J. van Wijk. Squarified Treemaps.

[12] S. Bruckner, S. Miksch, H. Pfister, Arlind Nocaj, and Ulrik Br. Computing voronoi treemaps faster, simpler, and resolution-independent.

[13] F. Aurenhammer. Voronoi diagrams a survey of a fundamental geometric data structure. *ACM Computing Surveys (CSUR)*, 23(3):345–405, 1991.

[14] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag TELOS, Santa Clara, CA, USA, 3rd ed. edition, 2008. ISBN 3540779736, 9783540779735.

[15] B.A. Price, R. Baecker, and I.S. Small. A Principled Taxonomy of Software Visualization. *Journal of Visual Languages and Computing*, 4(3):211–266, 1993.

[16] BA Myers, R. Chandhok, and A. Sareen. Automatic data visualization for novice Pascal programmers. *Visual Languages, 1988., IEEE Workshop on*, pages 192–198, 1988.

[17] M.E. Tudoreanu. Designing effective program visualization tools for reducing user's cognitive effort. *Proceedings of the 2003 ACM symposium on Software visualization*, 2003.

[18] M. Petre, AF Blackwell, and TRG Green. Cognitive questions in software visualization. *Software Visualization: Programming as a Multi-Media Experience*, pages 453–480, 1998.

[19] JI Maletic, A. Marcus, and ML Collard. A task oriented view of software visualization. *Visualizing Software for Understanding and Analysis, 2002. Proceedings. First International Workshop on*, pages 32–40, 2002.

[20] SP Davies. Display-based problem solving strategies in computer programming. *Empirical Studies of Programmers, 6th Workshop*, pages 59–76, 1996.

[21] B.A. Myers. Taxonomies of Visual Programming and Program Visualization. *Journal of Visual Languages and Computing*, 1(1):97–123, 1990.

[22] JT Stasko and C. Patterson. Understanding and characterizing software visualization systems. *Visual Languages, 1992. Proceedings., 1992 IEEE Workshop on*, pages 3–10, 1992.

[23] G.C. Roman and K.C. Cox. A Taxonomy of Program Visualization Systems. *Computer*, 26(12):11–24, 1993.

[24] Danny Holten. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE Transactions on Visualization and Computer Graphics*, 12 (5):741–748, 2006. ISSN 1077-2626. doi: http://dx.doi.org/10.1109/TVCG.2006.147.

[25] Ben Shneiderman. Tree visualization with tree-maps: 2-d space-filling approach. *ACM Trans. Graph.*, 11(1):92–99, 1992. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/102377.115768.

[26] B. Shneiderman and M. Wattenberg. Ordered treemap layouts. *Information Visualization, 2001. INFOVIS 2001. IEEE Symposium on*, pages 73–78, 2001.

[27] M. Wattenberg. Visualizing the stock market. *Conference on Human Factors in Computing Systems*, pages 188–189, 1999.

[28] JJ Van Wijk and H. Van de Wetering. Cushion treemaps: visualization of hierarchical information. *Information Visualization, 1999.(Info Vis' 99) Proceedings. 1999 IEEE Symposium on*, pages 73–78, 1999.

[29] Georgy Fedoseevich Voronoy. URL http://www-history.mcs.st-andrews.ac.uk/Biographies/Voronoy.html.

[30] S. Fortune. A sweepline algorithm for Voronoi diagrams. *Algorithmica*, 2(1):153–174, 1987.

[31] Q. Du, V. Faber, and M. Gunzburger. Centroidal Voronoi Tessellations: Applications and Algorithms. *SIAM Review*, 41(4):637–676, 1999.

[32] Henry S. Warren. *Hacker's Delight*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. ISBN 0201914654.

[33] Steven Raemaekers; Arie van Deursen; Joost Visser;. The maven dependency dataset, 2013. URL http://dx.doi.org/10.4121/uuid:68a0e837-4fda-407a-949e-a159546e67b6.