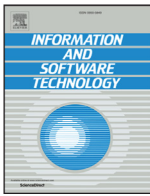
Information and Software Technology 87 (2017) 242–258

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof



Stable and predictable Voronoi treemaps for software quality monitoring

Rinse van Hees*, Jurriaan Hage*

Department of Information and Computing Sciences, Utrecht University, Princetonplein 5, Utrecht 3584 CC, The Netherlands

ARTICLE INFO

Article history:
Received 2 February 2016
Revised 1 July 2016
Accepted 23 October 2016
Available online 16 November 2016

Keywords:
Software visualization
Voronoi diagrams
Hilbert curves
Software quality monitoring
Stable Voronoi treemaps

ABSTRACT

Context: Voronoi treemaps can be used to effectively visualize software quality attributes of a given software system. Algorithms for computing Voronoi treemaps are non-deterministic making them unsuited for monitoring the development of such attributes over time.
Objective: We adapt an existing sweep line algorithm to efficiently compute Voronoi treemaps and we introduce a novel algorithm that adds stability and predictability.
Method: We introduce stable and predictable Voronoi treemaps based on additively weighted power Voronoi diagrams. We employ scaled Hilbert curves to place Voronoi sites in the plane, retaining the order in which sites are placed along the curve for easy comparison with revisions of the same software system.
Results: Our algorithm achieves a predictable first good approximation of the final location of the sites. We show that our algorithm not only provides more stability, but also that because of better placement it needs fewer iterations to compute its result. As part of our implementation we introduce a visualization to show the difference between two versions of a software system. We also present a small case study in which we use a web based application that implements our work to investigate the usefulness of stability and predictability of visualizations.
Conclusion: It is possible to achieve stable and predictable visualizations of software system attributes, while, as a pleasant side effect, decreasing the number of iterations necessary to arrive at the visualization.

© 2016 Elsevier B.V. All rights reserved.

1. Introduction

Many current software visualizations are geared towards a one-off 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 user 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/her insight instead of having to create it anew.

Voronoi treemaps are well suited for visualizing attributes of large software systems. In Fig. 1, we depict a version of Apache Jackrabbit Core [1] as generated by our tool. Here each region (at the deepest level) represents a method, and higher levels represent the class and package structure. The size of the Voronoi regions is determined by the *lines of code* attribute, and color is determined by the method's *McCabe's cyclomatic complexity*. The initial placement of vertices (along a Hilbert curve) is governed by yet a third metric. In all, our pictures incorporate information from three possibly distinct metrics. The picture clearly shows that the code base contains many packages of low complexity, and one package of high complexity.

Voronoi treemaps use Voronoi tessellation to build very readable and aesthetically pleasing pictorial representations of hierarchical data. They are well suited for visualizing software systems, because these systems often possess such a structure, e.g., inheritance hierarchies and package structures. However, the original Voronoi treemap algorithm is not suitable for visualizing different versions of the same software system, because it generates wildly different visualizations. In Fig. 2, we illustrate the non-deterministic nature of the standard algorithm with two visualizations of the *exact same system* as generated by the same implementation. The problem is even more noticeable when (slightly) different versions of the same system are visualized.

What is needed is an algorithm that is, in a sense, continuous (or non-chaotic): small changes to the structure of a software system and its attributes should lead to only small changes in the

* Corresponding authors.
E-mail addresses: rvanhees@pubpres.net (R. van Hees), J.Hage@uu.nl (J. Hage).

http://dx.doi.org/10.1016/j.infsof.2016.10.003
0950-5849/© 2016 Elsevier B.V. All rights reserved.

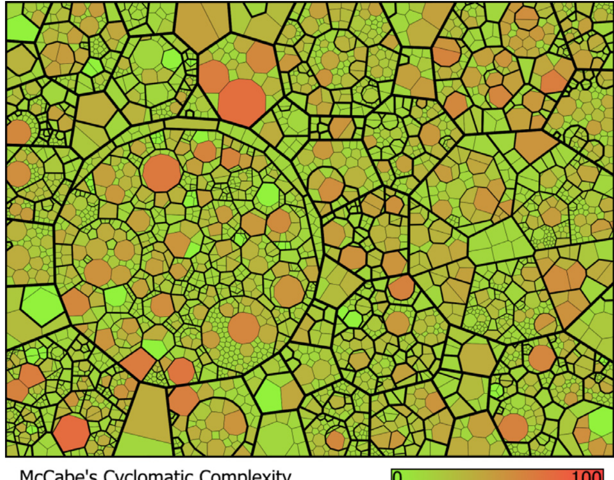


Fig. 1. A Voronoi treemap for Apache Jackrabbit Core 1.4.5.

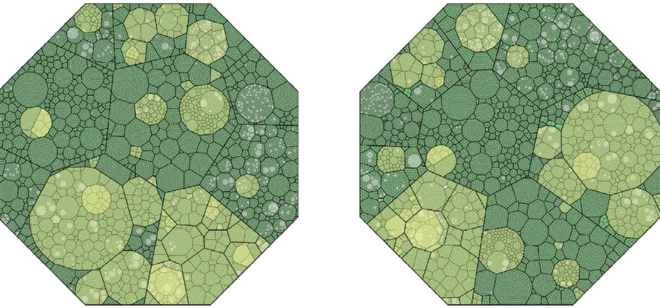


Fig. 2. Comparison of two Voronoi treemaps of the same system.

Voronoi treemap visualization; in particular, if the software system is unchanged, the algorithm should generate the exact same visualization. In this paper we show how to extend the original Voronoi treemap algorithm to produce visualizations in such a fashion.

Our visualization technique has the following characteristics:

- It produces pictorial representations of the structure and attributes of large software systems, allowing the viewer to better understand the software system, and monitor the changes over time.
- It is deterministic, producing the same pictorial representation for multiple runs on the same input data.
- It is versatile enough to allow the creation of pictorial representations of different attributes of the same software system with up to three different attributes in one picture.
- Pictorial representations of slightly different versions of the same software system should lead to only slightly different visualizations, so that dissimilarities should be easily identifiable. Large differences, on the other hand, lead to large differences in the pictorial representations.
- It is able to create the pictorial representation with reasonable speed, and needs fewer iterations than previously known algorithms.

In particular, we introduce stable Voronoi treemaps based on two distinct 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 [2] extended with weighted Voronoi sites.

To create stable Voronoi treemaps we develop an algorithm based on Hilbert curves. We employ a strict order on the data, chosen by the user, that is to be visualized and use Hilbert curves to initially place the data points on the visualization in this chosen order. We find that points placed in this fashion are much closer to the final positions as the algorithm converges. This both decreases the number of iterations of the algorithm to attain convergence, and also makes point placement more predictable.

This paper is an extended and modified version of our conference paper [3]. In Section 5.2 we introduce a new algorithm that improves upon our original algorithm. The new algorithm needs fewer iterations to achieve a more stable and predictable result. We include this new algorithm in our comparison of speed, stability and predictability in Section 5.3. A completely new addition is the small case study that we discuss in Section 7. The case study was done at a large software development company and uses a web application that implements our algorithms. The same web application is used to produce the pictures in this paper. Many of the pictures are the same as in our original paper because the new algorithm produces very similar pictures as the original algorithm. We have used some of the additional space to show new pictures produced by our tool, in particular a set of pictures for a range of versions of a software system and the differences between those versions. We also present four pictures that show different visualizations of the same system based on different sets of metrics.

The paper is organized as follows. We introduce general related work on software visualization in Section 2. Our work builds upon Voronoi diagrams, so we devote a special section to background information on Voronoi tessellation: Section 3. We discuss additively weighted power Voronoi diagrams and provide our novel algorithm for computing them in Section 4. In Section 5 we consider how to place the initial locations of the Voronoi sites along a Hilbert curve. We discuss the freely available implementation (in Java) of our work in Section 6. In Section 7 we discuss a small informal case study on the use of our implementation in a software development company. Section 8 concludes this paper.

2. Related work

Balzer et al. [4] introduce a variation on traditional treemaps that is more suited for software visualization: Voronoi treemaps. Software systems usually have a rich hierarchical structure; twenty levels deep is not unthinkable. Traditional treemaps are then not so suited, resulting in thin, elongated rectangles with a high aspect ratio. Using general polygons instead of plain rectangles can solve this issue. Using Voronoi tessellation to divide the available space for the top-level and recursing into the other levels leads to treemaps that are easier to read. Like us, Hahn et al. [5] discuss a method to achieve stable Voronoi diagrams, using a deterministic initial distribution that reduces the variation in node positioning if changes occur in the hierarchy. However, they employ the path in the source repository tree, which is subject to change if files are added, deleted or renamed. Moreover, in their placement they do not take the target weight of the node into account. Their algorithm is a variation on the work of Nocaj and Brandes [6]. According to the authors of [5] the implementations show the same speed and both are not ready for interactive use.

Steinbrückner's EvoStreets [7] uses the concept of city maps for visualizing aspects of software. He labels continuous understanding of evolving data sets with the term "consistency", and the topic of his thesis is to add consistency to software cities. As his extensive empirical study shows this comes at the price of lowering compactness. Note that Voronoi treemaps are 100% compact: all available space is used.

Hierarchies are widely used as models of the (static) structure of software systems. Visualizing these hierarchies poses some interesting problems though. For example, how do we position the nodes in two- or three-dimensional space? Noack and Lewerentz

[8] tackle this layout problem. For a variety of analyses of the static structure of software systems they derive the requirements for graph layouts that support those analyses. Because no single layout can satisfy all requirements, they introduce a space of layout styles. In this space, the layout styles are organized along the following three dimensions: degree of clustering, degree of hierarchicalness, and degree of distortion. By extending the minimization of energy function, a widely used method for computation of graph layouts, with these dimensions, they allow for the automatic computation of layouts for analyses.

The SHriMP (Simple Hierarchical Multi-Perspective) tool of Storey and Michaud [9] visualizes the static aspects of Java applications including Javadoc, package structure and source code. Operations such as zooming and filtering are supported to enhance the visualizations, while the visualizations themselves are largely based on graph layout algorithms.

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. The challenge is to ensure that the visualizations are easily comparable, meaning that objects should be in approximately the same place. Pinzger et al. [10] achieve this by superimposing the metrics and relations of multiple versions on a single Kiviat diagram.

Gîrba et al. [11] use historical information to show how a software system evolved. Since the history of a large software system can be very large, they developed ways to effectively summarize this information. This is very different from our work in which we visualize different snapshots of a system in a way that makes it easy to compare them pairwise, but does not try to capture multiple revisions in one picture. The visualizations themselves are so-called polymetric views [12]. A polymetric view is, like the name says, a view that incorporates information from multiple metrics. In our work, we can take at most three metrics into account (initial location along the Hilbert space curve, color and size). They can handle up to five metrics at once, in some cases only three. Their work has been implemented in the CodeCrawler tool.

The open visualization toolkit introduced in [13] 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. Later work by one of the authors [14] uses a pixel-oriented approach, in which 2D dense pixel orthogonal layouts are used to visualize various kinds of software entities: code in a single source file and multiple versions of the same file. Textures, color and shading are used to visualize all kinds of aspects of the software, both attributes and structure. An important quality aspect for their work was scalability, while at the same time keeping things simple and easy to learn.

EVolve [15] 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. 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.

Recent work on software visualization, particularly for the evolution of software systems, includes [16] which discusses a tool for visualizing software clones using hierarchical edge bundles. In this work, removed elements are retained, reducing compactness particularly when a long evolution is visualized. Also, [17] discusses evolution at a rather high level. Visual animations are employed in [18] to show how a software system evolves over time. The visualizations are meant to be easily explored. A complication that arises is how to improve coherence: if conceptually small changes happen to the system, how does one guarantee that the visualization also changes little? In other words, they tackle the same problem as we do. The work discussed in [19] also has many commonalities with our own, in the sense that they seek to achieve small changes in the visualization when the underlying data only changes little. In their paper, they discuss a new layout approach (based on software cities) which explicitly takes the development history of software systems into account, increasing stability across different revisions, but also makes the revision history itself clearly visible in the visualizations. Note that none of these works use Voronoi treemaps.

There also exists a large body of work that visualizes dynamic aspects of a system, i.e., its execution. This in contrast with our own work that looks at the dynamics of the structure of the software. For example, Cornelissen et al. [20] developed the SDR framework to visualize JUnit test execution traces, that can be shown at different levels of abstraction. Dynamic execution traces can generally be useful to understand the internals of a complex software system. Holten [21] introduced hierarchical edge bundles to visualize the large and many traces of a software system in a manageable way. The Java Interactive Visualization Environment (JIVE) [22] is an online visualization and analysis system for Java supporting both forward and reverse execution, and graphical queries over program execution. Recent work [23], combines the dynamics of execution and the dynamics of software evolution within a single framework, based on the VERSO visualization framework for constructing heat maps.

3. Background: Voronoi diagrams and Voronoi treemaps

3.1. History

In the early 1990's, Ben Shneiderman wanted to create a compact visualization of directory structures showing where his precious hard disk space went. He decided to divide his computer screen into rectangles, in alternating horizontal and vertical directions as you descend down the directory tree. The size of the rectangles reflected the size of the directories or files on the file system. This solution made optimal use of the limited screen space available in the 1990's. The first publication discussing this algorithm was published in 1992 [24]. The original algorithm has one shortcoming, one that most extensions inherit: it divides the space for each step in one dimension only. 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 is hard for users.

To alleviate the problem, [25] introduced ordered treemaps. Squarified treemaps, an evolution of ordered treemaps. use a different division algorithm, resulting in rectangles that have an aspect ratio close to one [26]. This division algorithm places in order of size which means that the original order of the data is always lost.

A remaining problem is that of hierarchy visualization: in the original treemap implementation hierarchy was reasonably well visible, by alternating the horizontal and vertical division at each level. In the new algorithms this was lost, and hierarchy became hard to discern. Van Wijk and Van de Wetering [27] therefore introduced cushioned treemaps, adding shading to each rectangle in such a way that it is easier to distinguish the parent-child relationships in the tree.

Voronoi treemaps are another way of solving both the space division problem and the hierarchy visibility problem [4]. Instead of using rectangles, Voronoi treemaps use polygons to divide the

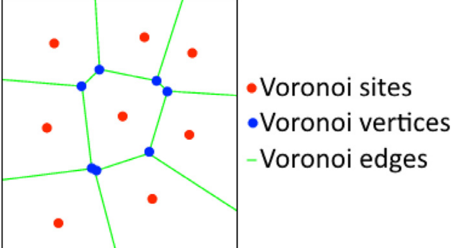


Fig. 3. Voronoi diagram.

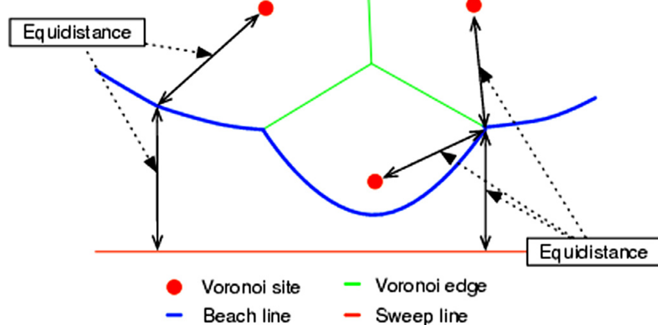


Fig. 4. Fortune's sweep line algorithm.

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 approaches one. This makes it easy to compare the size of objects. Another big advantage is that the Voronoi treemap division algorithm does not enforce an order in which nodes need to be visualized. This means that we can decide on the order ourselves and use this to enhance our visualizations. They do have one major drawback: because the algorithm is based on a random initialization, each visualization is very different. This means that in its current form Voronoi treemaps cannot be used to visualize the same data set multiple times. We also cannot use these Voronoi treemaps to visualize different versions of the same system, see Fig. 2.

3.2. Algorithms and terminology

To make sense of our later discussion, we introduce some of the terminology pertaining to Voronoi tessellation and treemaps, and provide some high-level details of the algorithms involved. More details can be found in [28].

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. Formally, consider a set of *sites* $S\{s_1, s_2, \ldots, s_n\}$, with each s_i a (distinct) point in the plane. The *Voronoi diagram* of S is the partitioning of the plane into n Voronoi regions and their edges (illustrated in Fig. 3). The (Voronoi) region of site s_i consists of all points $q \in \mathbb{R}^2$ where

$$dist(q, s_i) < dist(q, s_j) \text{ for each } s_j \in S \text{ with } j \neq i \,.$$

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 a (Voronoi) edge that is equidistant from s_i and s_j. This edge is a segment of the perpendicular bisector of the line segment $\overline{s_i s_j}$. An 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*.

Fortune introduced an efficient sweep line algorithm for computing the Voronoi diagram of a set of points [2]. Conceptually it uses a horizontal sweep line that traces out the Voronoi edges for a given set of points as it moves from top to bottom. In addition to the sweep line, the algorithm continually updates what is called the *beach line*. These two lines divide the plane into three distinct parts (see Fig. 4): everything above the beach line has been fully computed and can no longer change, everything below the sweep line is not yet visible to the algorithm, and everything in between the two is still in limbo. The beach line is equidistant from the site that is bounds and the sweep line, tracing out the curve as displayed in Fig. 4.

While the sweep line moves from top to bottom conceptually there are only two events that change the beach line and therefore influence the Voronoi diagram. The first of these events is the site event (see Fig. 5): when the sweep line encounters a site a new beach arc is introduced in the beach line which adds a new edge in the Voronoi diagram. The second is the circle event (see Fig. 6): when the sweep line touches the lowest point of a circle that is tangent to three sites, a beach arc is removed from the beach line. At this point, a beach arc shrinks to a point and its two adjacent beach arcs will meet. This corresponds to a vertex in the Voronoi tessellations. A more detailed explanation is given in [28, Section 3.2].

4. Creating Voronoi treemaps

In this section we give a novel algorithm for creating Voronoi treemaps based on additively weighted power Voronoi diagrams (AWP Voronoi diagrams for short) [30]. AWP Voronoi diagrams differ from traditional Voronoi diagrams in that the Voronoi sites have an associated weight. Weights can be used to encode yet another metric into a single Voronoi diagram; in most of the examples in this paper generated by our tool, we use weights to represent the *lines of code* attribute. In Section 4.1 we introduce our novel sweep line algorithm to compute AWP Voronoi diagrams. In Section 4.2 we show how AWP Voronoi diagrams can be used to create Voronoi treemaps.

4.1. Additively weighted power Voronoi sweep line algorithm

Our novel algorithm for computing AWP Voronoi diagrams is based on Fortune's sweep line algorithm as presented in Section 3.2. Here we extend Fortune's algorithm to take a weight associated with Voronoi sites into account by using the power distance function.

The power distance function for point p and site q with weight q_w is shown in Eq. (1).

$$dist(p, q, q_w) = (p_x - q_x)^2 + (p_y - q_y)^2 - q_w \quad (1)$$

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".

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

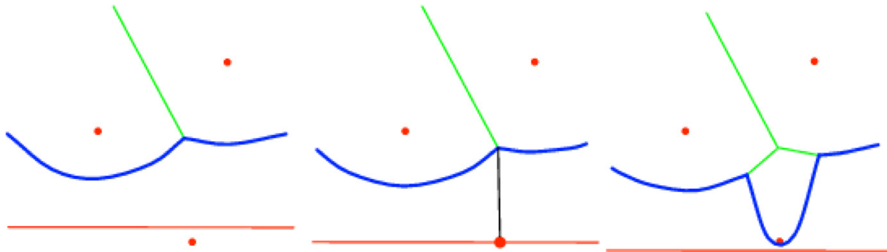


Fig. 5. Site event: a new arc is added to the beach line (adapted from [29]).

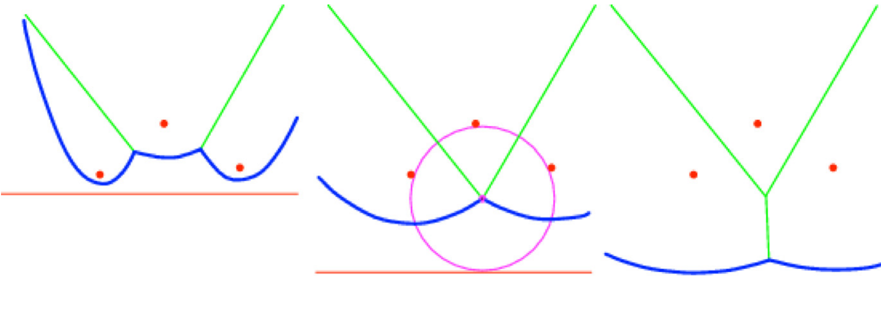


Fig. 6. Circle event: an arc is removed from the beach line (adapted from [29]).

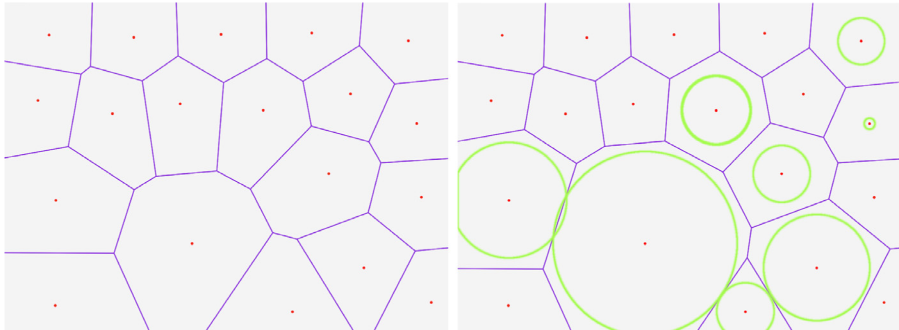


Fig. 7. Voronoi diagram (left) and AWP Voronoi diagram (right) of the same set of sites.

diagrams the Voronoi regions created using the power distance function are convex polygons. Fig. 7 shows a Voronoi diagram and an AWP Voronoi diagram for the same set of sites; the difference that the distance function and the weights (indicated here by green circles inside the rightmost picture) make is clearly visible.

Our algorithm for computing AWP Voronoi diagrams is a sweep line algorithm based on Fortune's algorithm, but there are major differences. When moving the sweep line over a set of weighted sites, and the sweep line touches the top of the corresponding circle, the power distance from the sweep line to the Voronoi site is zero. Originally we thought that this should introduce a new beach arc into the beach line, but that turned out to be wrong: Voronoi sites that are 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, and we simply insert a new beach arc into the beach line when the site itself is encountered. However, for circle events we have to diverge from the original algorithm. A new event and an updated sweep line are also necessary for our algorithm. We describe our extensions for the new algorithm in 4.1.1–4.1.3.

4.1.1. Updated circle event

Our treatment for circle events does change. Originally, they took place when three Voronoi sites were lying on the same circle. Now, we should consider the radical center of the three circles corresponding to three Voronoi sites. The radical center can easily be found by intersecting the radical axes of each pair of circles. The circle with its center at the radical center and orthogonal to all three circles, is now the circle for our circle event for AWP Voronoi diagrams. When the sweep line encounters the bottom of this circle, an arc disappears from the beach line.

4.1.2. New event: top site circle event

For our algorithm we need to introduce a third event. 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 mark the beach arc that lies directly above the site, because it is influenced by this site. We found that the marked arc should not be at equal power distance from the sweep line, but from this site. When one of two arcs is marked, we calculate the bisectors of this site and the sites corresponding to the arcs. 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

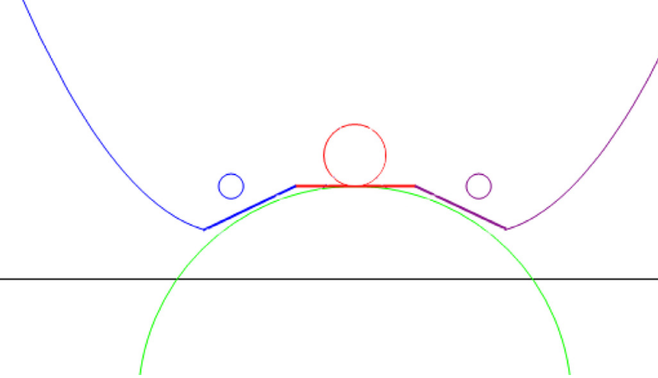


Fig. 8. Updated beach line with bisectors.

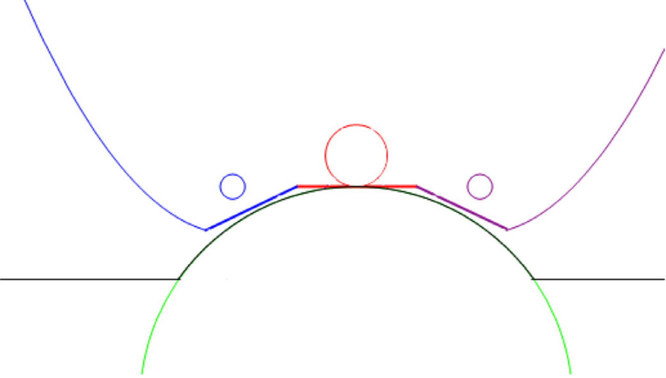


Fig. 9. Updated sweep line with a site that is yet to be encountered.

the sweep line, we use the normal intersection calculation. Otherwise, we use the intersection of the bisectors. Fig. 8 illustrates this updated beach line with bisectors.

4.1.3. Extended sweep line

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 lying within the circle of the site may happen. This means that we remove an arc from the beach line while this should in fact not yet happen. To solve this, we change the sweep line from a simple straight line to a line that "follows" the circles of sites that are yet to be encountered, see Fig. 9 for an illustration. 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. At this point, it may be that the parabolas are not correct below the sweep line, but as we make no claims about anything below the sweep line, this is not a problem. We forego a more detailed explanation of how our algorithm precisely handles these issues, but see [28] for more details.

4.2. Voronoi treemap algorithm

Having discussed how to construct AWP Voronoi diagrams, one piece is still missing: (recursively) building a Voronoi treemap. For each node in a graph we create an AWP Voronoi diagram that divides the Voronoi region of its parent. The root node has a Voronoi region equal to the area that the complete visualization has to fill.

In Voronoi treemaps the size of a Voronoi region represents a value of a (leaf) node in the data we are visualizing. Because we know what size a Voronoi region has to have to correctly represent the value we can create AWP Voronoi diagrams and iteratively change the weights of Voronoi sites to grow or shrink their region. To ensure that no two Voronoi regions overlap we also move the location of the Voronoi site in each iteration. The algorithm for recursively constructing Voronoi treemaps and iteratively creating AWP Voronoi diagrams was first described in [4]. It was later extended in [6,31]; in this paper we follow [6]. A part of the instability of Voronoi treemaps is caused by these algorithms: the movement of sites is hard to predict and between visualizations Voronoi sites can end up on very different locations.

4.3. Limitations

4.3.1. Additively weighted power Voronoi diagrams

Where Fortune's algorithm is proven to be an optimal solution for calculating Voronoi diagrams we have not tried to prove this for our algorithms. The additional complexity introduced by extending each step of Fortune's algorithm to deal with the power distance function and the weight of the Voronoi sites has a large impact on the complexity of the algorithm.

Especially the calculations for the intersections of beach arcs has become more computationally expensive. For each intersection we not only have to check the influence of the sweep line, but also the influence of all the Voronoi sites that have a corresponding circle that intersects with the sweep line. This is exacerbated by the fact that in our implementation circles of sites generally grow to only a little smaller than the Voronoi region. This means that there are many circles intersecting with the sweep line.

4.3.2. Voronoi treemap algorithm

The iterative algorithm used to create the AWP Voronoi diagrams for Voronoi treemaps converges to a reasonable solution, but still results in some Voronoi regions that are too large or too small. In Section 5.3 we will show that the algorithm converges to a lower bound, but never reaches the perfect solution. A more in-depth explanation on why there are sites that remain too big or too small can be found in [6].

Another limitation that is inherent to AWP Voronoi treemaps: when a smaller site is sandwiched between a large site and the edge of the polygon that is being divided, it is limited when increasing its size. It can only increase its size by pushing its other neighbors out, but those can only move when pushing their neighbors out. This means that it can take many iterations to reach a good approximate size, especially if the other neighbors are in the same situation. In Fig. 10 an example is shown.

5. Hilbert space filling curves for better initial site placement

We now consider how to place the initial locations of the Voronoi sites in order to obtain

- a more predictable placement, and
- a smaller number of iterations before the iterative site placement stabilizes.

The two are not unrelated: if you start out with a better placement, Voronoi sites will need to move around less before ending up in a convergent solution, making the placement more predictable.

Our goal is therefore to choose a deterministic and reasonable initial placement for Voronoi sites, and our solution to achieve this is by placing the Voronoi sites along a space-filling Hilbert curve. Fig. 11 shows three superimposed Hilbert curves of order 1 (dark), 2 (lighter) and 3 (light). We start by dividing up a square area into

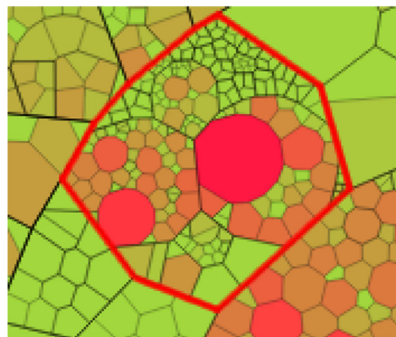
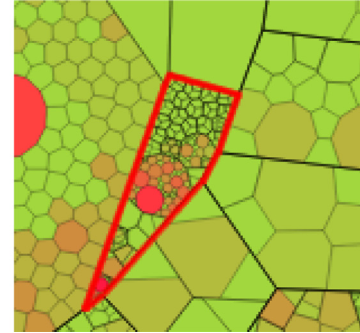



Fig. 10. Two versions of the same module visualized at the same zoom level. On the right it is sandwiched between a very large neighbor and an edge, on the left this is not the case and the correct size is shown.

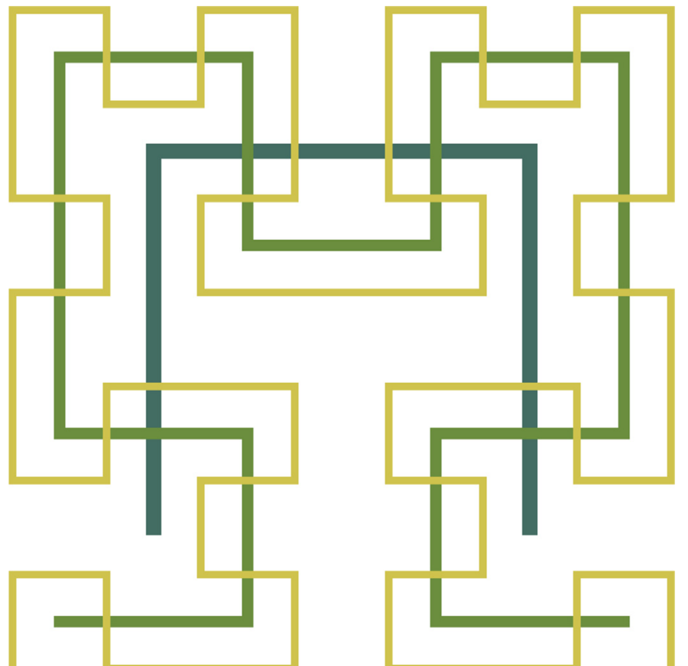


Fig. 11. Three superimposed Hilbert curves of order 1, 2 and 3 (reproduced from [33]).

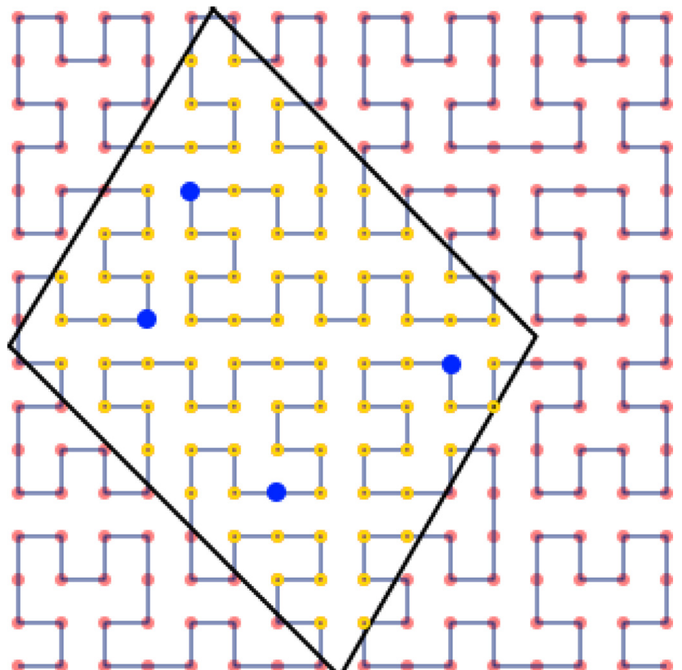


Fig. 12. A Hilbert curve intersected with a polygon and four Voronoi sites placed at equal intervals.

so called *cells*, these cells themselves are also squares. For a Hilbert curve of order 1, the area is divided up into four equal sized cells, and each time we increase the order by 1, all cells are similarly subdivided. So the grid of cells for a Hilbert curve of order 3, consists of 8 by 8 cells. The Hilbert curve is then 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. In our implementation, the user can elect the order in which Voronoi sites are placed on the curve. The same placement will then be used for revisions of the same software system, guaranteeing that the more similar revisions are, the more likely the resulting pictures are visually comparable, providing the necessary stability to our algorithms.

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 aligning the Hilbert square with the oriented axis-aligned bounding box 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. 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 Fig. 12.

In this paper, we show Hilbert curves of at most order 4, but in our actual implementation we also use higher order Hilbert curves. 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.

5.1. Variably spaced Voronoi sites

Although the above serves to illustrate the general idea, it can be improved. Instead of placing the points uniformly, we decided to take the desired area of the Voronoi region into account. In other words: if a Voronoi site has a large weight, we place it further away from its predecessor along the curve. This simple idea is, however, still not good enough. The Hilbert curve has a tendency of turning back on itself, so that points later on the curve can accidentally be only a small distance from points very early on the curve. This means that if we place a Voronoi site

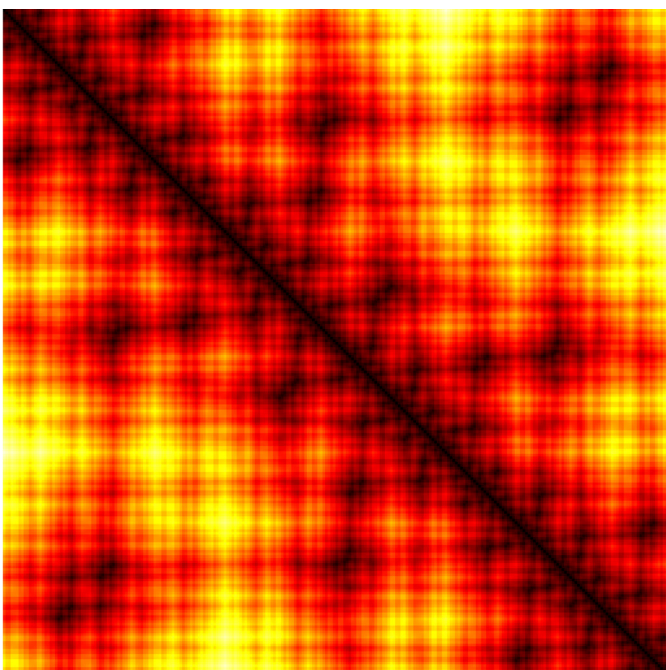


Fig. 13. A heat map of the Euclidean distance between any two points on a Hilbert curve of order 8 (reproduced from [34]).

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. Fig. 12 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 Fig. 13. 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.

We solve this problem by using one of four different Hilbert curves instead of the same one for every site. 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 Fig. 11, we can see that the lower the order of a Hilbert curve, the more guaranteed space each point has. It is okay to use different order Hilbert curves for different sites, because the general location of points in relation to each other is preserved. In this new variation of our 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. We chose these orders because large sites are the most troublesome to place with enough room to grow. The orders 1, 2 and 3 ensure that the sites with the largest desired region size will have enough room. For all other sites we use an Hilbert curve of order 8 to ensure that we have enough points available. 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 takes up. We then use the chosen curve to find a suitable point that lies within the bounding polygon. If we cannot find such a point we try a higher order curve. Once we reach the eighth order we keep trying until we find a suitable point. This gives us the algorithm as described in Algorithm 1 . We should

```
Data:
  The polygon P that has to contain the Voronoi sites
  The node r with associated metric m that is represented by P
  A hierarchical graph H with nodes with associated metric m
Result: A set S = {s_1, ..., s_j} of Voronoi sites
begin
    N ← ordered child nodes of r in H;
    S ← empty collection of Voronoi sites;
    B ← the bounding perfect square of P;
    areaFraction ← P_area / B_area;
    i ← 0;
    foreach n ∈ N do
        fraction ← n_m / r_m × areaFraction;
        if fraction <= 0.05 then
            (p, i) ← FindPoint on Hilbert curve of order 8 for n
            at i;
        end
        if fraction > 0.05 and fraction <= 0.10 then
            (p, i) ← FindPoint on Hilbert curve of order 3 for n
            at i;
        end
        if fraction > 0.10 and fraction <= 0.225 then
            (p, i) ← FindPoint on Hilbert curve of order 2 for n
            at i;
        end
        if fraction > 0.225 then
            (p, i) ← FindPoint on Hilbert curve of order 1 for n
            at i;
        end
        Create Voronoi site s at p;
        Add s to S;
    end
    return S;
end
```

Algorithm 1: Variably spaced Hilbert curve Voronoi site placement.

note that in our implementation we use a modified Hilbert curve of order 1 that has three additional points halfway in between the four existing ones to give us a little more choice when looking for a location for a new Voronoi site.

At this point, our algorithms do indeed start with pretty good locations for the Voronoi sites when creating a single level of the Voronoi treemap. By also giving a good starting weight we can do a bit better. Instead of starting our algorithms with sites that all have a weight of 1, we introduce an extra step 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 ensure that the circle of the site does not enclose another site.

5.2. Scaled Hilbert curve site placement

In our variable Hilbert curve site placement algorithm, we used just four orders Hilbert curves which meant that we almost always had to choose a Hilbert curve that did not ensure the right amount of distance to the neighboring sites. In cases of Voronoi

sites with a large desired region size they were most often too close together and in the case of Voronoi sites with small desired regions they were too far apart. This mismatch in desired region size and Hilbert curve order meant that there was more movement than necessary. We realized that we could use any order Hilbert curve by using the weight of a site and the distance between two consecutive points on a Hilbert curve. If we see the desired area size of a region as the area of a circle, we can compute the corresponding radius. We then use the distance between two consecutive points on Hilbert curves for different orders to find the one that has a distance just smaller than the radius. We use the Hilbert curve we found to try to find a good starting position. If we cannot find a good starting position we fall back to a higher order Hilbert curve until we do find a good starting position. In Algorithm 2 we show our solution that introduces scaled Hilbert curves.

Data:
The polygon P that has to contain the Voronoi sites
The node r with associated metric m that is represented by P
A hierarchical graph H with nodes with associated metric m
Result: A set $S = \{s_1, \ldots, s_j\}$ of Voronoi sites
begin
  $N \leftarrow$ ordered child nodes of r in H;
  $S \leftarrow$ empty collection of Voronoi sites;
  $B \leftarrow$ the bounding perfect square of P;
  $areaFraction \leftarrow \frac{P_{area}}{B_{area}}$;
  $totalFraction \leftarrow 0$;
  foreach $n \in N$ **do**
    $fraction \leftarrow \frac{n_m}{r_m} \cdot areaFraction$;
    $radius \leftarrow \sqrt{\frac{fraction}{\pi}}$;
    $p \leftarrow null$;
    $hilbertCurveOrder \leftarrow \log_2\left(\frac{1}{radius}\right)$;
    while $p == null$ **do**
      $(p, totalFraction) \leftarrow$ find point on curve, see Algorithm 3;
      $hilbertCurveOrder \leftarrow hilbertCurveOrder + 1$;
    end
    Create Voronoi site s at p;
    Add s to S;
  end
  return S;
end

Algorithm 2: Scaled Hilbert curve Voronoi site placement.

With this new algorithm we found that we not only improved the starting positions of Voronoi sites, but we also found that we improved the predictability. In Algorithm 1 we still had to change the weight of Voronoi sites a lot during the first few iterations. During these weight changes, sites move in relation to the location and weight of their neighbors. Both weight and location can change and cause a site to move in an unpredictable manner. By minimizing the amount of weight changes we also minimize the movement of Voronoi sites, which means that Voronoi sites stay closer to their initial location.

Using scaled Hilbert curve site placement and computing good starting weights does improve the running time of the Voronoi treemap algorithm, but we found that we could still improve upon the starting position of the Voronoi sites. We noticed that sites that were close to a border of the bounding polygon had a smaller size region, during the first few iterations of the algorithm, than we expected with the starting weight they had. If we look at the weight of a Voronoi site as a circle with a radius of $\sqrt{weight}$, we can clearly see in Fig. 14 that a large part of the circle of a site that is close to a border lies outside of the bounding polygon. This explains why the starting location could still be improved upon. By making sure that the initial location of a site is far enough from a border, by computing a smaller bounding polygon, we ensure that we reserve enough room for the Voronoi region. For each order Hilbert curve we use a different size bounding polygon. In Algorithm 3 we give our implementation.

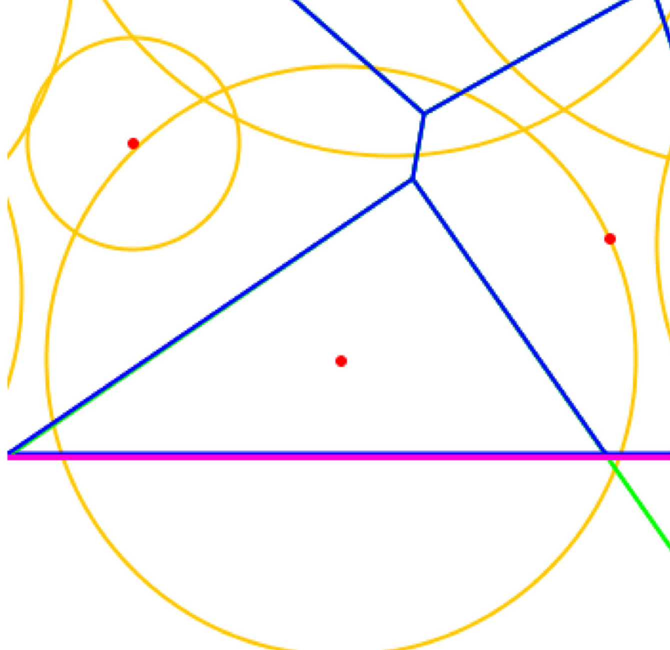


Fig. 14. A site that has a smaller area because it lies too close to the border (purple) of the bounding polygon.

5.3. Performance and stability

To verify the performance and stability of our algorithms we recreated the test setup from Nocaj and Brandes [6]. In their test they measured the total difference in desired area and actual area of the Voronoi regions. The test was repeated 250 times, using 50 Voronoi sites placed in a bounding box with an aspect ratio of 2: 1. We used the original Voronoi treemap algorithm as benchmark for our own algorithms. The results we obtained for our benchmark are similar to the results obtained by Nocaj and Brandes (note that we do not have their numbers, only what they displayed in a figure in their paper, so an exact comparison is not possible).

We tested both the variable spaced Hilbert curve algorithm from Section 5.1 (from the conference paper [3]) and the scaled Hilbert curve algorithm from Section 5.2 that is new in this paper. In Fig. 15 the total area error is shown for all three algorithms. We can clearly see that both our algorithms reach a lower bound area error in less iterations than the original algorithm does. We can also see that the scaled Hilbert curve algorithm is an improvement on the variable spaced Hilbert curve algorithm.

Using the same test setup we also compared the movement of sites in each iteration. In Fig. 16 we show the comparison of the total movement between the three algorithms for every second iteration. We can clearly see that using random starting positions produces much more movement than using variable or scaled Hilbert curves. Even when using variable Hilbert curves and a good initial starting weight there is still a lot of movement during the first few iterations. This is due to the fact that small sites start with an area that is too large and large sites start with an area that is too small. This means that all sites move a little. With the scaled

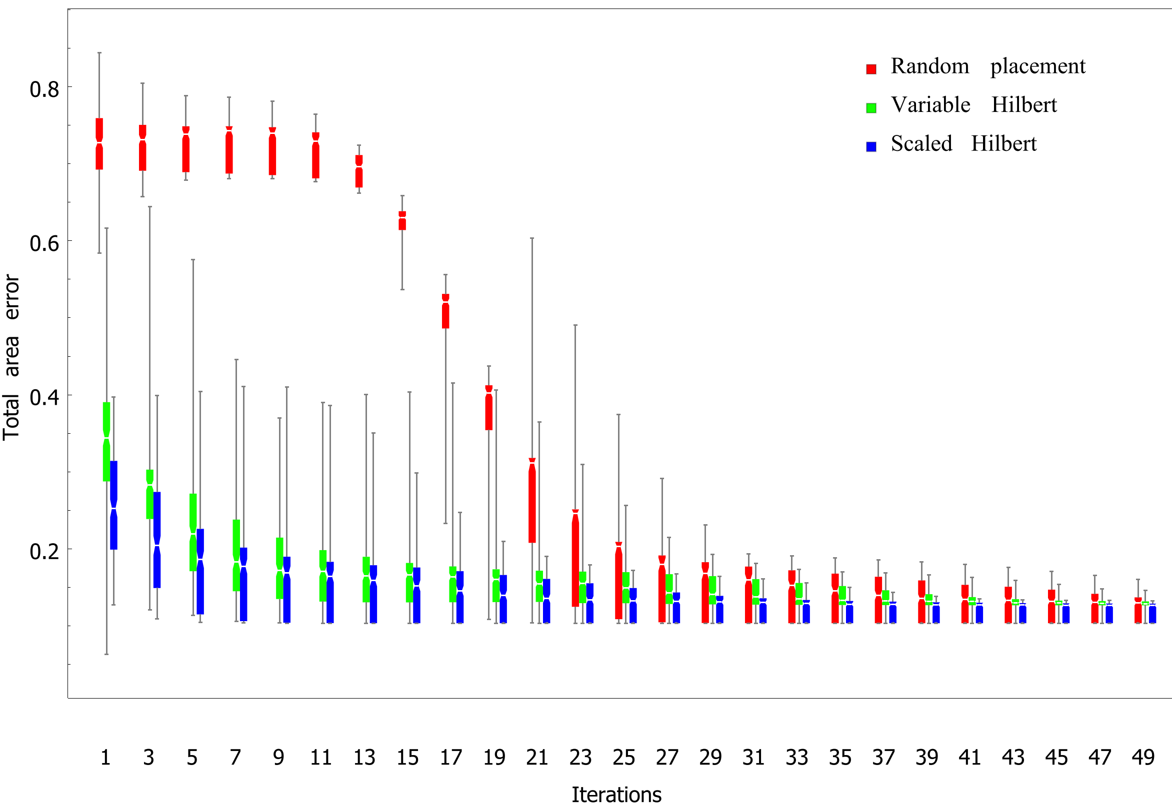


Fig. 15. Boxplot showing the total area error for every second iteration. Visualized 250 sample instances with random ordered sites, 50 sites placed in a rectangle with aspect ratio 2: 1, and target areas drawn from a power-law distribution with $f(x) = \frac{1}{x^4}$.

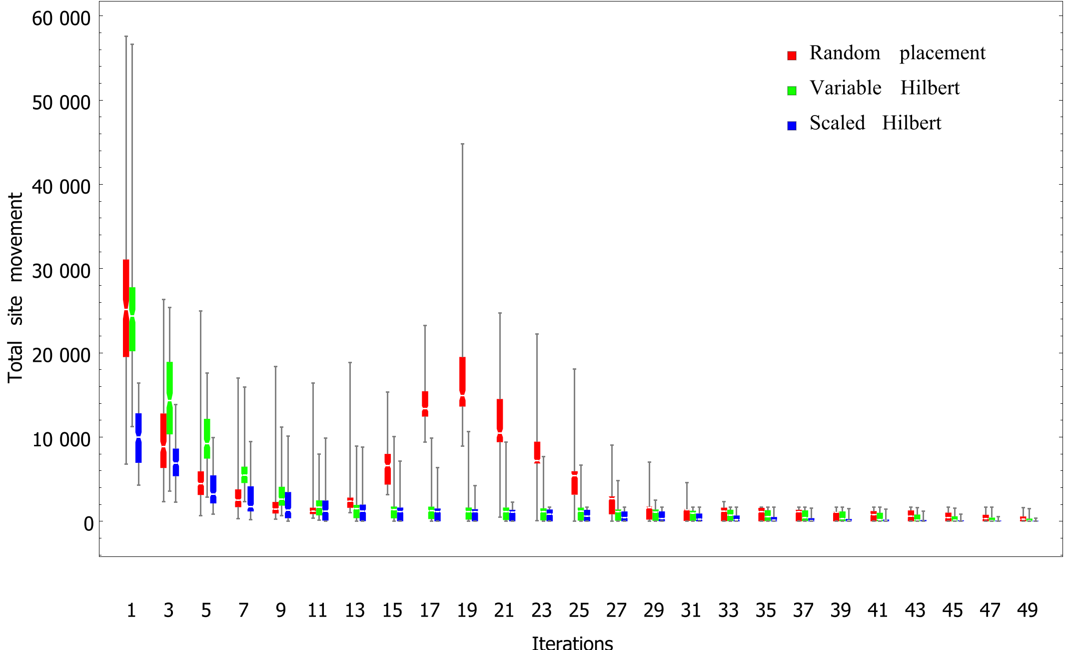


Fig. 16. Boxplot showing the total movement of all sites every second iteration. Visualized 250 sample instances with random ordered sites, 50 sites placed in a rectangle with aspect ratio 2: 1, and target areas drawn from a power-law distribution with $f(x) = \frac{1}{x^4}$.

Hilbert curve algorithm there is less movement, resulting in a more predictable final location of the Voronoi sites.

For random and non-scaled Hilbert curve site placement we cannot compute a better starting weight, because sites are often too close together to give a good relative weight. As a result, the first few iterations are spent moving sites to a more even distribution. When a more or less even distribution is reached the weights start to have an impact and we see more movement until a stable point is reached. We conclude that a good starting weight reduces movement and we can only use a good starting weight when using scaled Hilbert curve initial site placement.

Data:
totalFraction
areaFraction
radius
hilbertCurveOrder
boundingPolygon
Result: Point p that lies within *boundingPolygon* or null
totalFraction updated to the current location on the Hilbert curve
begin
  $lineSectionLength \leftarrow \frac{1}{2^{hilbertCurveOrder}}$;
  $index \leftarrow totalFraction \cdot 4^{hilbertCurveOrder} + \frac{radius}{lineSectionLength}$;
  testPolygon ← create new polygon by moving the borders of *boundingPolygon* *lineSectionLength* closer to the center of weight;
  $tries \leftarrow \frac{1}{areaFraction}$;
  if $lineSectionLength < radius$ **then**
    $tries \leftarrow \frac{radius}{lineSectionLength} \cdot tries$;
  end
  while $tries > 0$ **do**
    p ← find point at *index* on Hilbert curve of order *hilbertCurveOrder*;
    if *p lies in testPolygon* **then**
      return $(p, \frac{index + \frac{radius}{lineSectionLength}}{4^{hilbertCurveOrder}})$;
    end
    $index \leftarrow index + 1$;
    $tries \leftarrow tries - 1$;
  end
  return $(null, totalFraction)$;
end

Algorithm 3: Algorithm to find a point on a Hilbert curve.

5.4. Limitations

When creating a stable Voronoi treemap we have to keep in mind that Voronoi sites are ordered using the value of an attribute of the nodes. If this value changes between versions this will have the visual representation as the deletion of a node and the introduction of a new node. If the values of the attributes do not have a strict order we cannot guarantee stability; even when visualizing the same version the visualizations will be different.

When visually comparing Voronoi treemaps of different versions and their differences in the underlying data (Table 1), we can clearly see that the stability introduced by our algorithm maintains the changes between visualizations in relation to the changes in the underlying data. However, we have not formally proven that this property always holds.

6. Implementation

To be able to inspect the visualizations obtained from our optimized stable Voronoi treemap algorithm, as described in Section 5.2, an application with a web interface was built. The application allows a user to interactively create visualizations. To allow the users to get a complete overview of the visualizations, several example cases are provided. In this section an overview of the web interface and the example cases are given. The implementation is available for download from http://foswiki.cs.uu.nl/foswiki/Hage/Downloads.

The software systems we use were taken from a large online repository that stores both source code and a number of source code complexity measures [35]. For each method in each source file in the data set various pieces of information are stored including McCabe's cyclomatic complexity, LOC, the number of arguments to the method, and its label, i.e., the path from the root of the package to the source file and method.

6.1. Application

We have built an application with a web interface to verify that our algorithms in fact produce reasonable results. To actually create Voronoi treemaps, the implementation needs to be provided with software analysis results; the application organizes these analysis results by software project and release. For a given software system, multiple versions of the system may be present, and the web application allows for the easy comparison of these versions.

The algorithms should be supplied with a hierarchy in which to each node a number of values is associated; the web application itself is not able to compute a graph and metrics. The web application has a facility for uploading this information in the GEXF format (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 hierarchies. Also, attributes can be defined and added to the nodes at will. These two constructs provide the algorithms with enough of the right structure to create Voronoi treemaps.

6.2. Creating a Voronoi treemap

After uploading the data for a given software system revision, the user needs to select a number of attributes. First, there is the attribute that governs the relative size of the Voronoi regions (in our examples we use LOC). Second is the attribute that governs the order in which nodes in the graph are placed on the Hilbert curve (in our examples we use the fully qualified class name attribute). Clearly, the availability of these depends on what attributes are provided in the uploaded data set. By default, the colors of the Voronoi treemap represent *McCabe's cyclomatic complexity* of each leaf node. This can be changed by selecting another attribute in the color attribute drop-down.

The treemap is rendered as a Scalable Vector Graphics (SVG) picture, implying that zooming in does not result in artifacts and loss of detail. Fig. 17 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. Clicking on an area will display the attributes with their values.

6.3. Comparing Voronoi treemaps

The application also supports an easy comparison of two revisions of the same system. The difference with the ordinary visualization lies in the use of colors: the color of the leaf nodes now represents the difference between the two Voronoi treemaps for a selected attribute. We make a distinction between comparing a revision with a previous and with a next revision.

When comparing a revision with a previous revision we highlight the new leaf nodes in green. The intensity of blue shows by how much the value of the selected attribute has decreased since the previous revision. The intensity of red shows the increase of the value of the selected attribute since the previous revision.

When comparing a revision with a future revision we highlight the leaf nodes that will disappear in the future revision in purple. The nodes that are colored blue have an attribute with a value that decreases in the future revision. Nodes that are colored red have an attribute with a value that will increase in the future revision.

Table 1
Several versions of Apache Jackrabbit Core visualized. The differences in the attribute used to compute Voronoi region are also shown.

Version No.	Visualization	old vs new	new vs old
1.4.5			
1.4.9			
1.5.0			
1.5.3			
1.6.0			
1.6.2			
1.6.4			
2.0.0-BETA6			

The intensity of the color indicates the amount of change; white means no change at all.

In Fig. 18 we show the differences between two versions of Apache Jackrabbit Core. Fig. 18 shows the difference between version 1.5.3 and the future revision 1.6.0. In Fig. 18 we show the reverse: the difference between version 1.6.0 and the previous revision 1.5.3.

6.4. 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 90,000 lines of code, Apache Jackrabbit Core is a large software system, and with over 35 snapshots in the data set [35], a good fit for our valida-

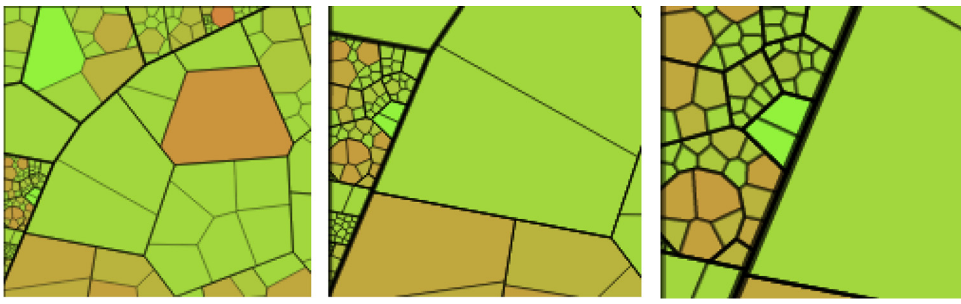
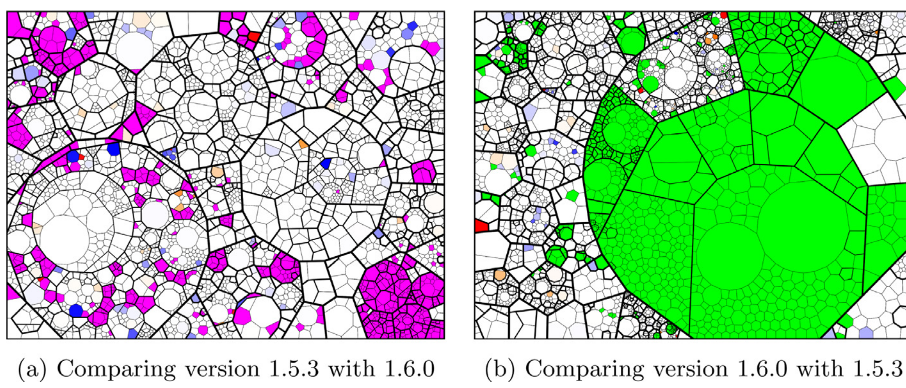


Fig. 17. Voronoi treemap at several zoom levels.



(a) Comparing version 1.5.3 with 1.6.0 (b) Comparing version 1.6.0 with 1.5.3

Fig. 18. The difference between Apache Jackrabbit version 1.5.3 and 1.6.0.

tion. The snapshots range from version 1.0.0 to 2.2.7 and there is a significant amount of change between those versions.

In Table 1 we show visualizations of eight versions of Apache Jackrabbit Core and the differences between consecutive versions. The value of the attribute Lines of Code (LOC) is used to compute the size of the Voronoi regions and the attribute value for the McCabe's complexity is visualized by the color. We show both the difference between a version and the next version, and the difference between a version and the previous version (as described in Section 6.3.)

The Voronoi treemaps in Table 1 clearly show that the stability introduced by our algorithms keeps the amount of changes between visualizations in relation to the amount of changes in the visualized data. The changes between versions 1.4.5, 1.4.9 and 1.5.0 are large, but we can still recognize the different Java packages and their location is still relatively the same. The amount of changes between versions 1.5.3 and 1.6.0 are very large, because in version 1.6.0 Apache Jackrabbit Core introduces a large set of generated Java classes (the large green area in the fourth column). The generated classes replace functionality that originally was located all over the software system (the purple classes in the third column).

The difference between the visualization of versions 1.6.2 and 1.6.4 is larger than the difference in the data would indicate. This can happen when a site is almost directly below one of its neighbors and switches sides. If it previously was on the right side it would be pushed left and in the new situation it is pushed to the right. This is especially noticeable when the neighbor site is very large and sandwiches the site between the neighbor and a border. Even though the change is larger than expected we can still relate all the classes between the two versions.

6.5. Performance

We have used the web application we build to visualize many software systems. With approximately 270,000 lines of code Apache Jackrabbit is the largest system we have visualized. Computing a Voronoi treemap of a version of Apache Jackrabbit takes a little over 30 s on our five year old laptop with an Intel Core I7 processor and plenty of memory. We found this performance good enough to use the system for online exploration. While we are happy with the current performance, by parallelizing the computation of the different AWP Voronoi diagrams needed for one Voronoi treemap, we can easily improve the performance. We can also improve the performance of the comparison visualization which is very slow, around 20 s, for what it does. We reuse the Voronoi treemap that was used to visualize one of the versions we are comparing, so there is no cost incurred for creating the Voronoi treemap. But the calculation of the difference between the two systems is done brute force. For each node in one graph we search for a node with the same id in the other graph and compare the selected attribute value. This could be improved.

7. A small informal case study

As part of our effort to validate the usefulness of stable Voronoi treemaps, we did a small comparative study. The study was conducted with ten volunteer employees of a large software development company. All the volunteers had at least five years of programming experience and an interest in software quality monitoring. They were also familiar with source code analysis and metrics. We recognize that this is a very small sample size and that we cannot draw definitive conclusions from the study. However, we do feel that the study is valuable because, as far as we know, there has been no other study on the value and usefulness of Voronoi treemaps.

As part of our study, there were two main aspects we wanted to consider. First, we wanted to know if Voronoi treemaps have any benefit over software quality visualizations that are readily available today. Second, are stability and predictability useful improvements on traditional Voronoi treemaps? To find the answers to these questions we used the tool described in Section 6.

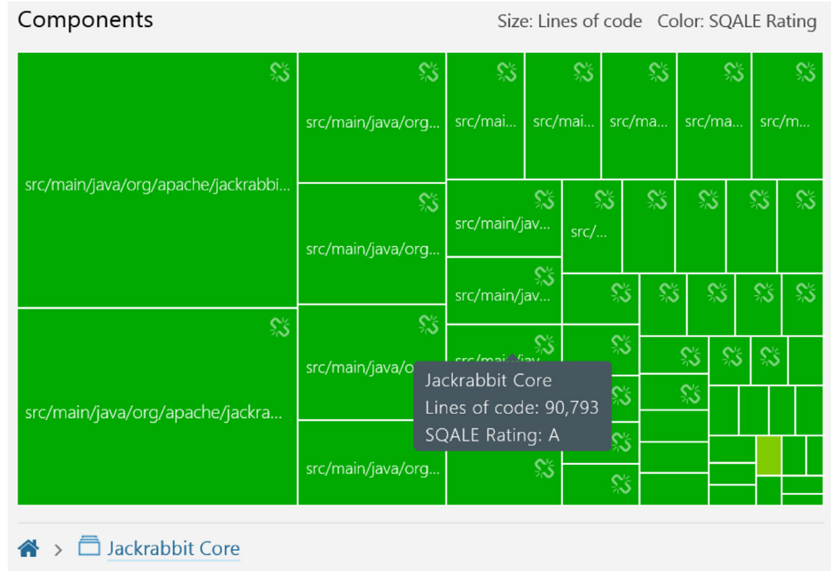


Fig. 19. The SonarQube treemap view for Apache Jackrabbit Core (reproduced from [37]).

Our study was done by providing each volunteer a computer with the required software installed and asking them to find answers to the following list of questions.

- Which five classes have the worst quality?
- Which package has relatively the most classes with low quality?
- Can you find an explanation for the larger number of low quality classes in this package?
- What is your opinion on the overall quality of the system? Give a grade between 1 and 10.
- If you were asked to improve the quality of the software, where would you start?

We observed how they came to the answer for each question and how long it took to find it. After all volunteers had answered all questions, we discussed the results in a group setting to evaluate the given answers, and to discuss any differences.

7.1. The value of Voronoi treemaps

To find out whether Voronoi treemaps have any benefit over software quality visualizations that exist today, we elected to compare Voronoi treemaps against the visualizations available in SonarQube. SonarQube [36] is a widely used open source project that provides a platform for source code analysis and presenting the results. SonarQube was familiar to all of the participants: they all used the web interface to monitor the quality of their software. In SonarQube most of the analysis results are presented as simple lists and tables, but SonarQube does offer a few visualizations. For example, SonarQube offers a squarified treemap with the aggregate of several quality metrics. The SonarQube project has a public demo instance of their software where they publish the analysis results of several open source programs. One of the analyzed programs is Apache Jackrabbit [37], in Fig. 19 the Apache Jackrabbit Core treemap view is shown. Because we also have analysis results for Apache Jackrabbit that can be visualized with our tool we decided to do our comparison using Apache Jackrabbit. We have chosen to use the analysis results for version 2.0.0 as we had the analysis results for that version available in both tools.

We split the group of volunteers into two equal sized groups. One group would be using SonarQube and the other group would be using our tool to answer the five questions listed above. For each question we measured the time it took to find the answer. We also discussed the effort it took to find the answers and asked for an overall experience of using either SonarQube or our tool. We discussed the results with all volunteers as a group.

During the case study we found that the time needed to answer each question varied widely within the two groups, which is why we decided that the timing was of little to no value. We therefore only focused on the answers and the conclusions that were drawn.

The five classes with low quality that were found largely overlapped between all the volunteers. The users of SonarQube had an easy time: they could use a sorted list with the worst offenders at the top. This sorted list used the aggregate of several metrics. The users of our tool had it somewhat harder: they had to create visualizations with the metrics that they believed best show the quality of a class. Most users of our tool used the visualization of *number of lines* as size and *McCabe's complexity* as color. One user opted to use *number of parameters* as size and *McCabe's complexity* as color. The consensus was that a class with many methods with high complexity is bad.

When trying to find the package that has the most classes with low quality in relation to the total number of classes in the package, the SonarQube users had a harder time, because all the lists show either only a single package or the complete system. When looking at the complete system it is hard to find all classes belonging to a package. When looking at a single package it is hard to compare it to other packages in the system. Using our tool this was easier, even though you still had to count classes by hand. The visualizations easily showed the worst offender and a quick count verified it.

The explanation for the larger number of low quality classes was easily found. The name of the package gave it away, having "generated", "xml" and "parser" in its name. The package contains generated code for an XML parser and the complexity of parsing XML documents results in complex code that is hard to understand.

The question about the overall quality of the system is where we found the first major difference in opinion between the two groups of volunteers. We asked both groups to not take the generated code into account when evaluating the quality of the whole system. The group that was using SonarQube had to use the aggregated overview page, the squarified treemap with little detail or had to go into the detailed listings. This meant that they

focused on those classes that were reported as having a lower quality, but could not directly relate them to the complete system. While SonarQube itself reported the overall quality as good, an A grade, the amount of warnings and low quality classes that were reported gave a different impression. The volunteers using SonarQube gave a grade between 6 and 7.5. The volunteers that used our tool could clearly see how the different subsystems related to each other and could reason about the quality of the system as a whole. They gave the system grades between 8 and 8.5. The users of our tool did not give the overall quality as high a grade as SonarQube itself did; they noticed several classes and packages that have a high number of functions with higher than average complexity. They felt that while each function on itself might not be very complex, when they are grouped together it indicates a complex piece of software that might be hard to reason about.

Clearly the two groups had a different view on the overall quality of the system as a whole; the users of our tool found the overall quality higher than the SonarQube users, but lower than the quality grade reported by SonarQube. At this point we gave both groups access to both tools and asked them to re-evaluate their quality grade. What we found interesting to see was that with the visualizations provided by our tool, the volunteers could clearly pinpoint classes that needed further inspection and used SonarQube to navigate to the source code. After giving the volunteers some more time to play around with both tools, we asked them to grade the overall quality of the system again. The volunteers that originally used SonarQube now all gave a grade of 8 and the volunteers using our tool gave the same grade as they did before. We asked the volunteers that originally used SonarQube to explain why they gave a higher grade than before and they all agreed on the fact that they now could place the different metrics reported on classes in the context of the whole system. Before, they focused on the large number of reported low quality classes and warnings that SonarQube gave and could not get a feeling for the number of good quality classes. With the Voronoi treemap from our tool they could get a total view of the system. After having seen that SonarQube offers an aggregated quality score for classes, the users of our tool said that our tool could be improved by also offering an option to aggregate multiple metrics into a single metric for visualization.

The last question we asked was where they would start to improve the quality of the system. The answers between the two groups were largely in agreement. They both would pick the topmost offending classes and start there. There was one exception: a volunteer using our tool was of the opinion that a single low quality function or class in a package might be acceptable. The insight that this volunteer offered was that packages that have many classes and functions with a lower than average quality could very well indicate a problem that lies deeper than the lines of code themselves. It is very well possible this is an indicator of an architecture problem that could be solved on that level and increase the quality of the system significantly.

7.2. The value of stability and predictability

After evaluating the usefulness of Voronoi treemaps we also wanted to know if stability and predictability had an added benefit. We again split the group of ten volunteers into two groups of five. Both groups were given our tool, but only one group was given a fully functioning version of our tool. The other group we gave a crippled version of our tool: the ordering of the data that was being visualized was always random. Otherwise the tool was the same, which means that they still had all the other benefits of our tool, such as interactive creation of Voronoi treemaps, the ability to select one metric for size of a region and one for color, and the ability to compare two versions of the system. In Fig. 20 we show several example Voronoi treemaps, of the same version of Apache Jackrabbit Core, with different metrics for the available options.

For this part of the study we gave both groups the complete set of versions of Apache Jackrabbit we had source code analysis results for. Again, we formulated several questions for both groups to answer. Afterwards we discussed the results as a single group.

- How does the quality of class *org.apache.jackrabbit.core.WorkspaceImpl* evolve over time?
- In version 1.3.0 class *org.apache.jackrabbit.core.nodetype.NodeTypeDefDiff* is introduced. When looking at version 1.2.2, can you predict where the class will be in the visualization of 1.3.0?
- What is the big change that happens in the system between version 1.6.4 and 2.0.0?
- How has the quality of the system evolved over time?

When trying to answer the question about the quality of class *WorkspaceImpl*, it took the group that used the fully working version of our tool some time to find this class in the visualization, but then they quickly moved through the visualizations of several versions of the system. In each version they could quickly find the class, because it was visualized in approximately the same location. When a big change happened between two versions of the system, it took a little longer to find the class, but they often could quickly recognize the package and find the class in it. Because the order of classes in a package and functions in a class stays the same, it was easy to identify them even if the location was different. The *difference view* made it even easy to see small changes to the class. The group that was using the crippled version of our tool quickly gave up using visualizations and resorted to using SonarQube. Because in each visualization the location of packages, classes and functions was completely different, finding the class was hard. Using SonarQube they had to find the class in the listings, which was fairly easy using the search function. In the end, the amount of effort it took to come to the answer that the class had hardly changed was equal for both groups. While there was little change to the class itself, the users of the fully functioning tool could tell if the system as a whole was changing or not. The group that reverted to SonarQube could only talk about the single class and had no real notion of the changes in the overall system.

The question whether the volunteer could predict where a new class would show up in the visualization was somewhat unfair. The group using our crippled tool could not even come close; each time they visualized the system the outcome was completely different. The group using the full version of our tool could predict with high accuracy where the class was going to appear. Because there were few other changes between the versions, the location of the package that the class belonged to stayed the same. With the ordered placement of classes within a package they could even predict where the class would be visualized in the package.

The answer to the third question was fairly easy to find for both groups; a large part of the system was replaced with new generated code. For the group using our full tool this large change meant that the visualizations had a bigger difference than visualizations of any other two consecutive versions. Once one or two packages were identified, identifying the other packages was easy because their locations were predictable due to the predictable nature of our algorithm. Using the *difference view* it was easy to see which packages and classes were removed and which were new. The group that used the crippled version could still easily see that there was a big change. Finding which packages remained and which were gone was a much harder task. The *difference view* did make it possible, but they could not easily correlate packages between versions and some even resorted to making a list on paper

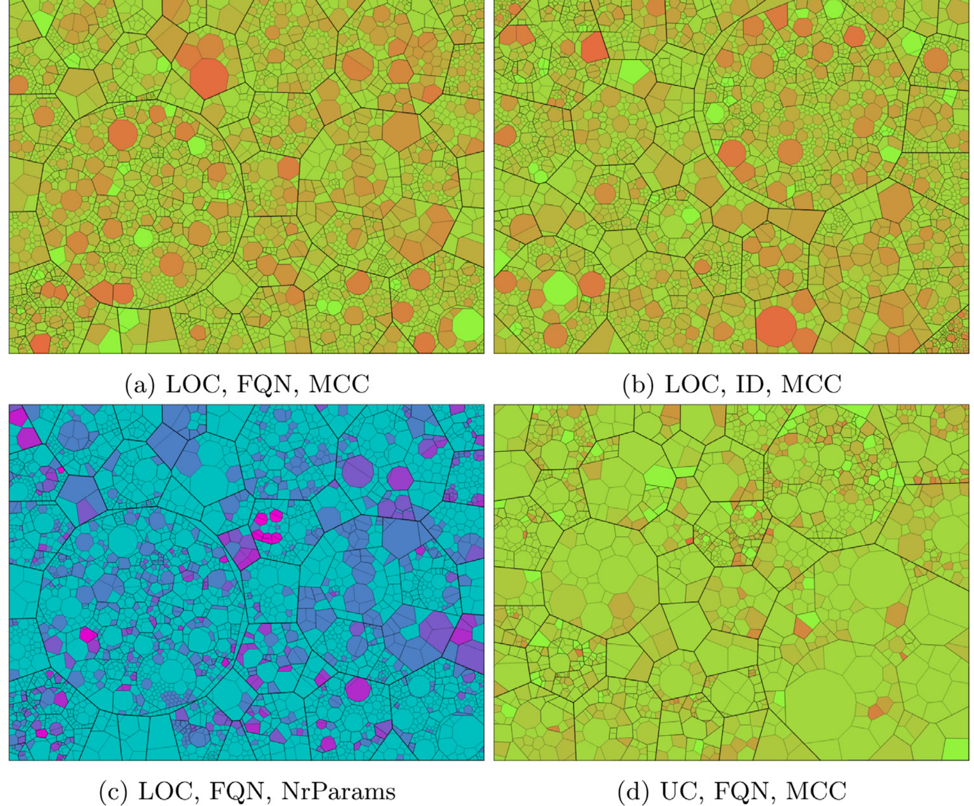


Fig. 20. Apache Jackrabbit Core 1.4.2 visualized using four different sets of metrics. The first metric is used to compute region size, the second is used to order the data, and the third is used to color the region. LOC: *Lines of Code*, FQN: *Fully Qualified Name*, MCC: *McCabe's Cyclomatic Complexity*, UC: *Usage Count*, ID: *unique IDentifier*.

and using that to keep track of what was new and what was removed.

The fourth question was about the changes in quality of the whole system over time. To help answer this question we prepared two slide shows: one using the full tool and one using the crippled version. In these slide shows, we showed a slide for each version with a Voronoi treemap with *lines of code* as size and *McCabe's complexity* as color. The slide show created with the full version of our tool showed a gradually and naturally changing software system that grows in size and complexity. The slide show made with the crippled tool was hard to interpret; each time you got a feeling for the quality of a version, the next Voronoi treemap was so different that it was hard to see what changed or stayed the same. The volunteers in the group that used the crippled tool did have a feeling that the system grew in size and complexity, but could not say with certainty if this was happening in a controlled and normal fashion.

7.3. Threats to validity

The case study was done with a small group of volunteers that already had an affinity with software quality monitoring, source code analysis and metrics. All volunteers were familiar with SonarQube and had used the web interface. None of the volunteers had any familiarity with Apache Jackrabbit which made placing the analysis results in context harder than with software they are intimately familiar with.

The case study was not set up as a formal case study; we had a discussion between the two main parts of the study in which the volunteers could influence each other. Also, several of the questions asked are biased in favor of stable Voronoi treemaps and we only used one software system when evaluating the usefulness.

The relation between software metrics and the quality of software is not always evident. The users of our tool only had access to visualizations of the raw metric values. While most agreed that a high *McCabe's complexity* is bad and that a high number of function parameters is also an indicator of harder to maintain source code, the metrics do not directly show low quality code. During our study we asked questions in relation to quality of a software system, but tried to answer those questions solely based on metrics. We had several discussions about how a certain metric relates to quality.

The functionality offered by SonarQube is much larger than the functionality offered by our tool. For example, SonarQube supports a much larger set of metrics and rules that can be measured. SonarQube also allows the user to click through to the source code. Our tool is strictly a visualization tool and is limited to the analysis results that are provided. The analysis results we used were much finer grained than those used in SonarQube; all metrics visualized in our tool were at the method/function level, while most of the metrics in SonarQube were at the class/file level.

7.4. Conclusion

Our study was too small to draw definitive conclusions, but we feel we can say that Voronoi treemaps are a valuable visualization when assessing a software system using metrics. Stability and predictability have as added benefit that insight gained from one version can be carried over to another version. The *difference view* of two versions helps in identifying exactly what changed, and because it uses the same stable Voronoi treemap the insight is transferable to the view of a single version of a software system.

The volunteers of our study suggested several improvements that we could add to our tool. While they had little to do with stable Voronoi treemaps as such, we do think they would be valuable additions. The suggestions as given by the volunteers are the following:

- A search function to find a package, class or function would help when looking for a specific package, class or function.
- A textual tree view of the system that highlights the node that is clicked on in the Voronoi treemap. This gives a more detailed view of a class or package.
- The ability to scale a Voronoi treemap in relation to its absolute metric size. When comparing two Voronoi treemaps the size of regions can be hard to compare, because the size is relative to the parent Voronoi treemap.
- The ability to click through to the source code. This enables the user to find out why a certain class or function has a certain metric value.
- The ability to combine two or more metrics to a new aggregate metric to use for either size or color in the created Voronoi treemap. Visualizing just two metrics does not always say enough about the quality of a system; visualizing more metrics at once could help.

8. Conclusion

In this paper we have shown how we are able to create stable and predictable Voronoi treemaps using scaled Hilbert curves and additively weighted power Voronoi diagrams. We explained our sweep line based algorithm for creating those additively weighted power Voronoi diagrams. We also showed how we evolved our usage of Hilbert curves from a simple deterministic site placement algorithm to a scaled approach that uses many orders of Hilbert curves to provide a higher level of stability and predictability. In the process we also improved upon the number of iterations it takes to create a Voronoi treemap, by selecting a better starting weight for Voronoi sites. Using a tool that implements our algorithms, we were able to show stable and predictable visualizations of multiple versions of a large software system and to do a small case study. The case study gives a first indication that stable and predictable Voronoi treemaps are valuable when looking at the quality of a software system evolving over time.

Topics for future work include a larger study on the usefulness of stable and predictable Voronoi treemaps, following the study of Steinbrückner [7]. We would also like to see if we can extend our algorithm to deal with relationships between visualized data items, so that related items end up close together. We suspect this may be possible by exploiting the order in which the visualized items are mapped onto the Hilbert curve.

Acknowledgments

The authors would like to thank Hans Bodlaender for his contribution to this project, and Joost Visser and others at the Software Improvement Group for initiating it. We also want to thank the anonymous reviewers for their many useful comments to improve this paper.

References

[1] Apache Software Foundation, Apache jackrabbit. URL http://jackrabbit.apache.org.
[2] S. Fortune, A sweepline algorithm for Voronoi diagrams, Algorithmica 2 (1) (1987) 153–174.
[3] R. van Hees, J. Hage, Stable voronoi-based visualizations for software quality monitoring, in: Software Visualization (VISSOFT), 2015 IEEE 3rd Working Conference on, IEEE, 2015, pp. 6–15.
[4] M. Balzer, O. Deussen, C. Lewerentz, Voronoi treemaps for the visualization of software metrics, in: Proceedings of the 2005 ACM Symposium on Software Visualization (SOFTVIS05), ACM Press, New York, NY, USA, 2005, pp. 165–172. http://doi.acm.org/10.1145/1056018.1056041.
[5] S. Hahn, J. Trümper, D. Moritz, J. Döllner, Visualization of varying hierarchies by stable layout of Voronoi treemaps, in: Proceedings of the 5th International Conference on Information Visualization Theory and Applications (IVAPP). SCITEPRESS-Science and Technology Publications, 2014.
[6] A. Nocaj, U. Brandes, Computing Voronoi treemaps: faster, simpler, and resolution-independent, Comput. Graph. Forum 31 (3pt1) (2012) 855–864.
[7] F. Steinbrückner, Consistent Software Cities: Supporting Comprehension of Evolving Software Systems, 2012. http://books.google.nl/books?id=GLPgngEACAAJ.
[8] A. Noack, C. Lewerentz, A space of layout styles for hierarchical graph models of software systems, in: Proceedings of the 2005 ACM Symposium on Software Visualization (SOFTVIS05), ACM Press New York, NY, USA, 2005, pp. 155–164.
[9] M.-A. Storey, J. Michaud, SHriMP views: an interactive environment for exploring multiple hierarchical views of a Java program, in: Proceedings of 9th International Workshop on Program Comprehension (IWPC01), 2001.
[10] M. Pinzger, H. Gall, M. Fischer, M. Lanza, Visualizing multiple evolution metrics, in: Proceedings of the 2005 ACM Symposium on Software Visualization (SOFTVIS05), ACM Press New York, NY, USA, 2005, pp. 67–75.
[11] T. Gîrba, M. Lanza, S. Ducasse, Characterizing the evolution of class hierarchies, in: Proceedings of the 9th European Conference on Software Maintenance and Reengineering (CSMR05), IEEE Computer Society, 2005, pp. 2–11.
[12] M. Lanza, S. Ducasse, Polymetric views - a lightweight visual approach to reverse engineering, IEEE Trans. Softw. Eng. 29 (9) (2003) 782–795.
[13] A. Telea, A. Maccari, C. Riva, An Open Visualization Toolkit for Reverse Architecting, in: Proceedings of the 10th International Workshop on Program Comprehension, IEEE Computer Society Washington, DC, USA, 2002.
[14] L. Voinea, A. Telea, Visual data mining and analysis of software repositories, Comput. Graph. 31 (3) (2007) 410–428.
[15] Q. Wang, W. Wang, R. Brown, K. Driesen, B. Dufour, L. Hendren, C. Verbrugge, EVolve: an open extensible software visualization framework, in: Proceedings of the 2003 ACM Symposium on Software Visualization, ACM Press New York, NY, USA, 2003.
[16] A. Hanjalic, Clonevol: visualizing software evolution with code clones, in: Proceedings of the First IEEE Working Conference on Software Visualization (VISSOFT13), 2013, pp. 1–4, doi:10.1109/VISSOFT.2013.6650525.
[17] R.G. Kula, C. De Roover, D. German, T. Ishio, K. Inoue, Visualizing the evolution of systems and their library dependencies, in: Proceedings of the Second IEEE Working Conference on Software Visualization (VISSOFT14), 2014, pp. 127–136, doi:10.1109/VISSOFT.2014.29.
[18] G. Langelier, H. Sahraoui, P. Poulin, Exploring the evolution of software quality with animated visualization, in: 2008 IEEE Symposium on Visual Languages and Human-Centric Computing, IEEE, 2008, pp. 13–20.
[19] F. Steinbrückner, C. Lewerentz, Understanding software evolution with software cities, Inf. Vis. 12 (2) (2013) 200–216. 1473871612438785
[20] B. Cornelissen, A. Van Deursen, L. Moonen, A. Zaidman, Visualizing testsuites to aid in software understanding, in: Proceedings of the 11th European Conference on Software Maintenance and Reengineering (CSMR07), IEEE, 2007, pp. 213–222.
[21] D. Holten, Hierarchical edge bundles: visualization of adjacency relations in hierarchical data, in: Proceedings of the IEEE Symposium on Information Visualization. IEEE Computer Society, 2006.
[22] P. Gestwicki, B. Jayaraman, Methodology and architecture of JIVE, in: Proceedings of the 2005 ACM Symposium on Software Visualization, ACM Press New York, NY, USA, 2005, pp. 95–104.
[23] O. Benomar, H. Sahraoui, P. Poulin, Visualizing software dynamicities with heat maps, in: Software Visualization (VISSOFT), 2013 First IEEE Working Conference on, IEEE, 2013, pp. 1–10.
[24] B. Shneiderman, Tree visualization with tree-maps: 2-d space-filling approach, ACM Trans. Graph. 11 (1) (1992) 92–99. http://doi.acm.org/10.1145/102377.115768.
[25] B. Shneiderman, M. Wattenberg, Ordered treemap layouts, in: Proceedings of the IEEE Symposium on Information Visualization (INFOVIS01), 2001, pp. 73–78.
[26] M. Bruls, K. Huizing, J.J. Van Wijk, Squarified Treemaps, Springer, 2000.
[27] J. Van Wijk, H. Van de Wetering, Cushion treemaps: visualization of hierarchical information, in: Proceedings of the IEEE Symposium on Information Visualization (INFOVIS99), 1999, pp. 73–78.
[28] R. van Hees, Stable Voronoi treemaps for software system visualization, http://www.cs.uu.nl/people/jur/msctheses/rinsevanhees-msc.pdf (2014).
[29] M.d. Berg, O. Cheong, M.v. Kreveld, M. Overmars, Computational Geometry: Algorithms and Applications, 3rd ed., Springer-Verlag TELOS, Santa Clara, CA, USA, 2008.
[30] M.-M. Deza, E. Deza, Dictionary of Distances, Elsevier, 2006.
[31] M. Wattenberg, Visualizing the stock market, in: Conference on Human Factors in Computing Systems, 1999, pp. 188–189.
[32] H.S. Warren, Hacker's Delight, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
[33] Wolfram Demonstrations Project, reproduced from http://demonstrations.wolfram.com/HilbertAndMooreFractalCurves/.
[34] Datagenetics, Hilbert curves, reproduced from http://datagenetics.com/blog/march22013/index.html.
[35] S. Raemaekers, A. van Deursen, J. Visser, The maven dependency dataset, (2013). 10.4121/uuid:68a0e837-4fda-407a-949e-a159546e67b6.
[36] SonarQube, SonarQube; an open platform to manage code quality. URL http://www.sonarqube.org/.
[37] SonarQube, SonarQube online demo instance for apache jackrabbit. URL https://nemo.sonarqube.org/overview?id=org.apache.jackrabbit:jackrabbit.