Voronoi Evolving Treemaps

Davide Paolo Tua, Roberto Minelli, Michele Lanza

CodeLounge @ Software Institute — USI, Lugano, Switzerland

***Abstract*—Since their invention, treemaps have been widely used to visualize hierarchical structures, due to their intuitive nature and their scaling capability: Indeed, given a maximum treemap size, one can depict hierarchical structures, such as file and software systems, of arbitrary size and depth. To make up for the rather blocky nature of treemaps, Voronoi treemaps were introduced, leading to depictions that look more "organic". However, hierarchical structures in general, evolve over time, and this is especially the case for software.**

We present Voronoi Evolving Treemaps (VET), a novel approach inspired by the Voronoi power-weighted treemap algorithm, that takes into account the evolution of hierarchical structures. VET is able to display the complete evolution of a software system in terms of its hierarchical structure, and enriches the visualization with additional information. We detail VET's evolutionary layout algorithm, discuss the architecture, implementation, and the features of VET, and illustrate how VET can be used to analyze the evolution of different systems.

***Index Terms*—software visualization, software evolution, treemap, layout, voronoi, power-weight**

I. Introduction

Hierarchical structures, commonly termed trees, are prevalent in software systems. They manifest themselves at the filesystem level (*i.e.,* directories which contain directories and files) and also at a programming language level (*i.e.,* packages contain classes which contain methods). Over the years two main ways have emerged to visualize such structures. The first one is the traditional graph approach where entities are nodes and containment is represented as edges. The other approach are treemaps, which have the advantage of being able to represent hierarchies of arbitrary size in a predefined space. However, the orthogonal nature of traditional treemap layouts often produces thin and stretched out rectangles and complicates the identification of boundaries between and within different hierarchy levels. Researchers tried to overcome these limitations in several ways, for example, by generating layouts where the shape of nodes is approximated to squares, as in *"Squarified Treemaps"* [1]. The latest development of treemaps, called *"Voronoi treemaps,"* use arbitrary polygons, instead of rectangles, to depict nodes and encode information on the nesting level in in border thickness between elements [2]. Figure 1 exemplifies how Voronoi treemaps can be used to visualize hierarchical information.

Although these approaches have improved the way hierarchical information is represented, they all share a major limitation when it comes to visualizing software systems: they do not deal with evolution. Software constantly evolves to adapt to changing requirement [3], thus software visualization approaches should consider evolution as first class concept.

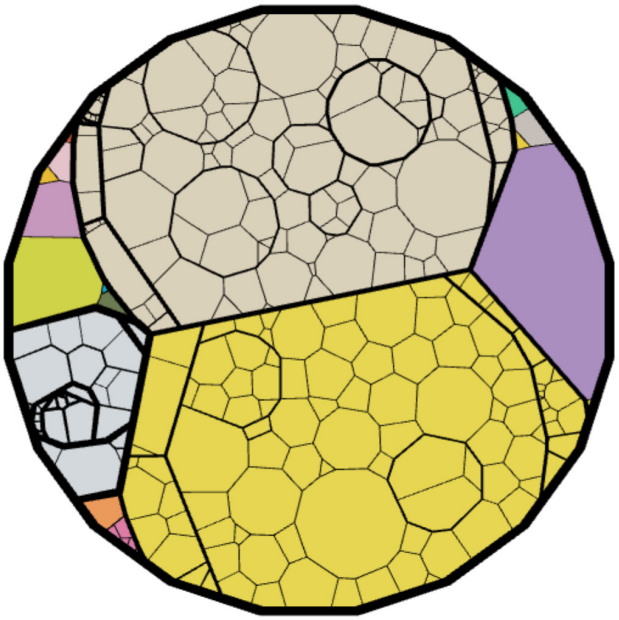


Fig. 1: A Voronoi Treemap of a Snapshot of a Software System

We introduce `VET`, an approach to *"Visualize Evolving Treemaps"* implemented in a publicly available web-application.[1] The layout of `VET` is based on Voronoi Power-Weighted Diagrams [4], [5] with a non-euclidean distance function. Each region of the treemap is proportional to an additive hierarchical metric of choice, *i.e.,* metrics for which the weight for a given node can be calculated as the sum of all its childrens' weights. If we use `VET` to visualize a software system, we can for example use Lines of Code (*i.e.,* LOC), code churn [6], or code complexity as weight metrics. Differently from most existing approaches, `VET` considers evolution as a first class concept. For each step S_i in the evolution of a system (*e.g.,* commits, time intervals, commit samples) `VET` generates a visualization. This visualization, will be used as starting point to generate the view for the next step S_{i+1}. This enables `VET` to maintain a consistent layout across versions. `VET` provides ready-made projects to visualize and enables users to analyze new Git repositories.

Structure of the Paper: Section II discusses Voronoi diagrams. Section III details our approach and Section IV exemplifies how to analyze software systems with `VET`. Section V summarizes the related work. Section VI concludes the paper.

[1] See **`https://vet.si.usi.ch/`**

II. Background: Voronoi Diagrams

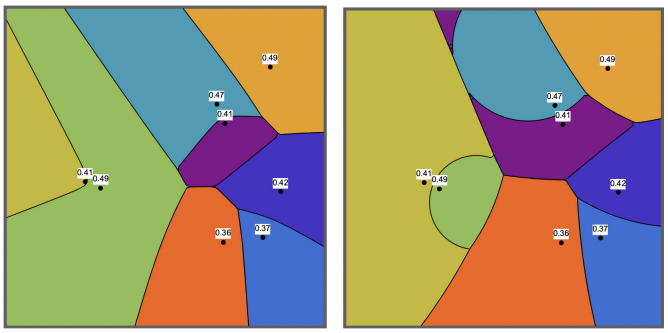
A Voronoi diagram [7] is *"a partitioning of a plane with n points (also called seeds, sites, or generators) into convex polygons (or regions) such that each polygon contains exactly one point and every point in a given polygon is closer to its generating point than to any other"* [8].

To define how "every point in a polygon is closer to its site than to any other" different distance functions can be employed. If the distance is euclidean, the resulting diagram will be composed of convex polygons, with the sites contained by their respective regions. If we add the constraint that the area of each Voronoi cell should be proportional to a specified value (*e.g.,* the value of a metric), we must use a distance function based on the *"weight"* of the site i.

The most used distance functions are:

1) ***Additive Weighted Distance*** where the weight is subtracted from the distance ($Da = D(p, p_i) - Fw_i$);
2) ***Multiplicative Weighted Distance*** where the distance is divided by the weight coefficient ($Dm = D(p, p_i)/Fw_i$);
3) ***Power-Weighted Distance*** where the weight is subtracted from the square of the distance ($Dp = D(p, p_i)^2 - Fw_i$).

Figure 2 shows the resulting Voronoi diagrams using (a) additive and (b) multiplicative weighted distance functions.



(a) (b)

Fig. 2: Voronoi Diagrams Using Different Distance Functions

The values of the metric (*i.e.,* w_i) and the respective weights in the diagram (*i.e.,* Fw_i) are two different values, related as follows: $Fw_i = f_i(w_i)$, where f_i is an unknown monotonically increasing function, different for each site i, that grows in a non-linear, non-predictable way. The problem of using such functions is that they often result in semantically wrong diagrams (*e.g.,* sites with no regions, regions disconnected from the site, non-straight region boundaries), like Figure 2a. Moreover, the initial position of sites often results in diagrams that are not appealing to the eye (*e.g.,* polygons with very high width to height ratio). For the first problem, the use of a power-weighted distance function mitigates at least some of these limitations by ensuring that all region boundaries are straight. To contrast the second problem and improve the look and feel of the diagrams, the computation is often followed by the execution of Lloyd's algorithm, also known as Voronoi relaxation [9], or by an analogous step resulting in a *CVT* (Centroidal Voronoi Tessellation).

III. VET: Visualizing Evolving Treemaps

A. Architecture

VET is a traditional web application composed of a backend and a frontend communicating through the WebSockets.[2] Figure 3 depicts the overall architecture of VET.

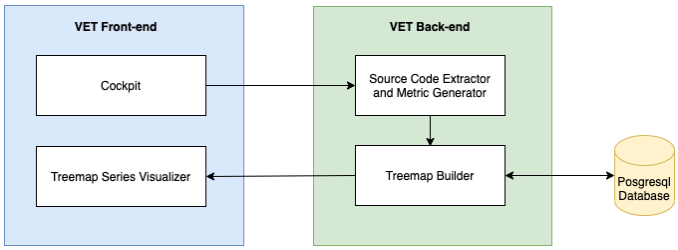


Fig. 3: The Architecture of VET

The ***backend***, written in Crystal,[3] is responsible for: (1) cloning the repositories (2) extracting the values of the metrics for every commit (3) performing the calculations to generate one Voronoi Power-Weighted treemap for each commit, and (4) sending the data to the frontend.

The ***frontend***, written in React, Typescript, and three.js, lets users specify the repository to analyze and to configure the parameters of the visualization.

B. The Evolutionary Algorithm

The underlying algorithm of VET has two main responsibilities: (1) extracting the values of hierarchical metrics across all revisions of the project, and (2) creating the Voronoi Power-Weighted treemap.

Extracting Hierarchical Metrics. Our algorithm clones the Git repository then sequentially scans all non-merge commits that were performed on the repository. For each commit, the algorithm extracts the value of the chosen hierarchic metric (*e.g.,* LOC) for each of the files in the repository. As a result, for each revision of the project we obtain a *metrics tree*. The whole process generates a list (L) of *metrics tree*.

Creating the Voronoi Power-weighted Treemap. For each node n in each *metrics tree* e of L, our algorithm computes a *Voronoi Cell Result* $V_{e,n}$ composed of (1) node name (2) site (3) Voronoi weight (fwi), and (4) polygon.

Algorithm 1: Evolving Treemap Generation

```
Input: list L of all metrics data, enclosing Polygon E
Output: list VTL of the resulting Voronoi treemaps
foreach element e in L do
    N ←nodes in e ;
    if e is not the first element then
        foreach node n in N do
            VTL_{i,n} ← (VTL_{i-1,n}) if exists VTL_{i-1,n};
        end
    end
    VTL_e ← MultiLayerVoronoi(N, rootNode, E)
end
```

[2]See `https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API`

[3]See `https://crystal-lang.org/`

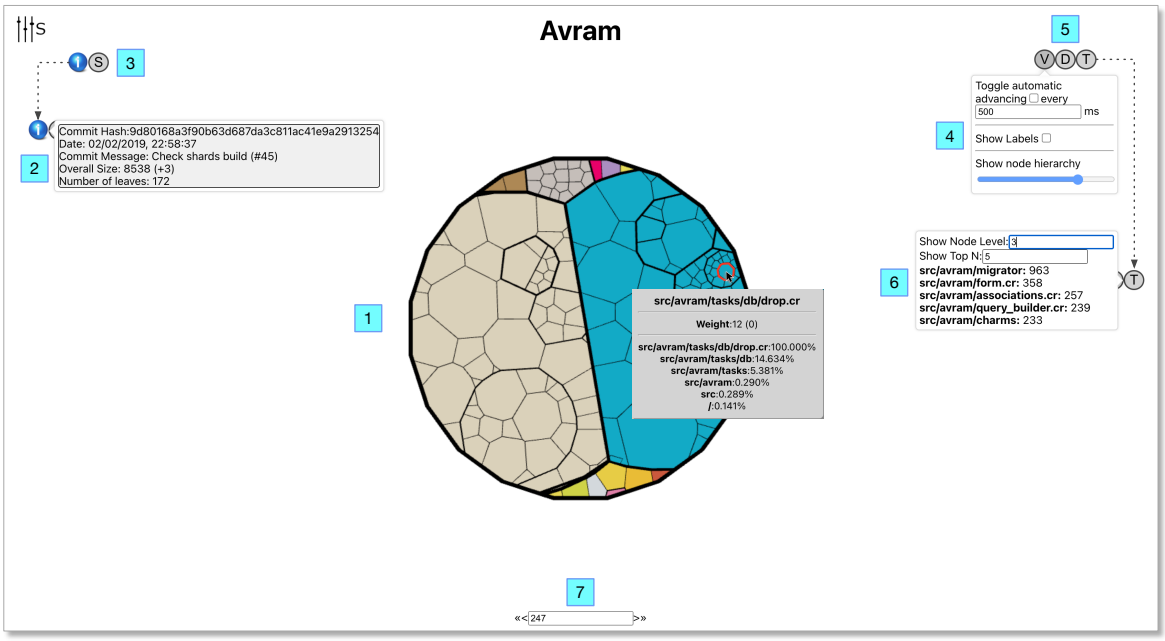


Fig. 4: The User Interface of VET

Algorithm 1 summarizes the iterative algorithm to build the whole list of *Voronoi Cell Result*.

Algorithm 2 recursively computes the single Voronoi treemap for any snapshot of the project.

Algorithm 2: MultiLayer Voronoi

Input: list N of nodes, starting node r, enclosingPolygon E
Output: list of outstructures VL
$C \leftarrow$childrenOf r;
if *C not empty* **then**
  $VL \leftarrow leftarrow SingleLayerVoronoi(C, E)$ **foreach** *child c in C* **do**
    $MultiLayerVoronoi(N, c, V.polygon)$
  end
end

Algorithm 3 computes a single Voronoi Power-Diagram.

Algorithm 3: SingleLayer Voronoi

Input: list N of nodes, enclosingPolygon E
Output: V for each e in N
$DA_i \leftarrow N_i.weight/N_i.parent.weight$ **foreach**) **do** (Initialization of new nodes
  node n in N
end
if *n does not have a pre-filled V structure* **then**
  V_n := (site: random point inside E,fwi: random float, polygon: nil)
end
while *stoppingConditionsNotMet* **do**
  $P_N \leftarrow VoronoiPowerDiagram(V_N.sites, V_n.weights)$
  foreach *P_i in P_N* **do**
    $P_{in} \leftarrow intersection(E, P_i)$;
    $A_i \leftarrow Area(P_{in}/Area(E))$;
    if $DA_i < A_i$ **then**
      $fwi = V_i.fwi + ErrorFunction(DA_i, A_i)$
    else
      $fwi = V_i.fwi - ErrorFunction(DA_i, A_i)$
    end
    $V_i = (centroid(P_{in}), fwi, P_{in})$
  end
  computeOverallAccuracy
end

This algorithm is inspired by Aurenhammer [4] and Nocaj [5]. It computes the Voronoi power-diagram for a set of $(site, weight)$ tuples by computing the corresponding convex hull and re-projecting it onto the plane, returning for each site i the corresponding polygon P_i. To ensure that the algorithm terminates, similar to the work of Nocaj, we implemented two types of stopping conditions: (1) accuracy (*i.e.,* average error is less than 5%) and (2) maximum number of iterations.

C. User Interfaces and Features

The GUI of VET is depicted in Figure 4, subdivided into:

1) Main Area: Shows the Voronoi Treemap representation of the code base at a given point in time. Hovering on a node provides information on its path, metric value compared to its parent chain and a link to that instant's repository file.

2) Metadata Popup: Provides information about the specific instant that is currently visualized. It shows for the currently visualized commit its hash, timestamp, the message, number of leaves and the overall metric weight.

3) Search by name: By providing a string, it highlights all the nodes whose name matches the string.

4) Visualization options: Three operations are provided to Autoadvance/Timelapse, show node labels, and select the maximum hierarchy node level to be displayed.

5) Delta Mode: A particular way of looking at the analysis provided by `VET` is not by considering the metric itself, but how the metric changes between epochs. We implemented a dedicated view called Delta Mode. It shows, using different colors, which nodes had a change in metric in the last N epochs, using progressive transparency (See Figure 5).

6) Top N nodes: This menu allows to look for the N nodes at a desired level that have the largest area (*i.e.,* the highest metric value). Hovering on any of those highlights them into the main area, to father their location.

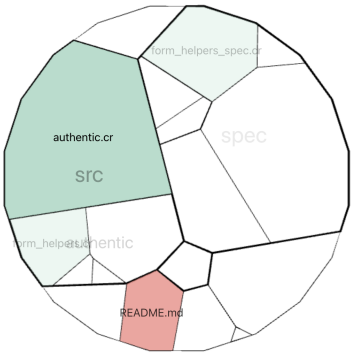


Fig. 5: An Example of Using the Delta Mode

7) Epoch selector: Provides a way to move across instants.

D. Workflow

Given a repository, the UI provides some options to generate the Voronoi Treemaps.

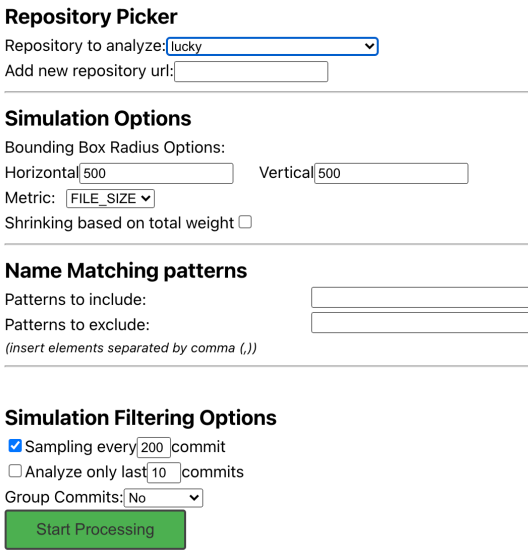


Fig. 6: The Settings of VET Used to Produce Figure 8

In particular, it provides (see Figure 6)

- a way to specify the size of the overall enclosing polygon;
- an option to set the previous size to be dependent of the overall weights (*e.g.,* a treemap with root weight equal to 2 is twice as big as one with root weight equal to 1);
- an option to choose the desired metric;
- a way to select (or reject) particular elements or subtrees of the repository;
- Options to sample the single commits by taking only the last N, by taking 1 every n commit or grouping them by day, week, or month.

When the backend completes its computation, it sends the result to the frontend which will show it like in Figure 4.

IV. Working Examples

We show the evolution of some GitHub projects, summarized in Table I.

TABLE I: Analyzed Projects in a Nutshell

Project Name	JetUML	Lucky	Lua
Creation Date (mm.dd.yyyy)	01.07.2015	01.06.2017	07.28.1993
No. Commits	2,044	1,018	5,381
No. Files (on Jun 18, 2021)	417	295	110
No. Contributors	19	76	4

A. JetUml

JetUML[4] is a desktop tool to design UML diagrams. For the repository analysis, we focused on two subfolders: `src` (*i.e.,* yellow in Figure 7) and `test` (*i.e.,* green in Figure 7). It can be seen how much the repository grew in size as the number of commits increased, and the increasing relevance of tests. However, in the third snapshot of Figure 7, the developer added a new package (*i.e.,* `org.json` highlighted in red), without creating the corresponding test folder.

B. Lucky

Lucky[5] is a web framework written in Crystal, with a very active community. This is highlighted by the growth of the codebase along time, as shown in Figure 8. We can also see that the balance between tests ('spec', in yellow) and source code ('src', in beige) remains constant over time.

C. LUA

Lua[6] is a programming language, especially used for mods or in embedding contexts. The repository has a different structure than the previous ones, with a very flat, unnested structure. As a result, the Voronoi Treemap has a more "harlequinesque" look (see Figure 9). Still, by using *delta mode*, it is easy to see what changed between any commit.

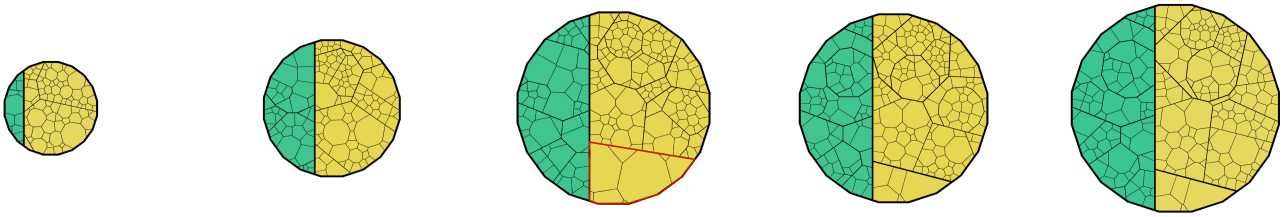
V. Related Work

Shneiderman [10] introduced the concept of treemaps to visualize the filled space on his hard disk. To prevent some of the visual artifacts from Shneiderman algorithm, Bruls *et al.* [1] proposed the squarified treemap approach.

Balzer *et al.* moved away from the traditional rectangle-based visualization and proposed using Voronoi diagrams instead [2], [11]. Nocaj *et al.* [5] proposed a new way of computing a Voronoi Treemap, abandoning the sample-based algorithm in favor of a resolution independent approach based on the work of Aurenhammer [4].

Differently from the aforementioned approaches, `VET` considers evolution a first class concept as, in most cases, the visualization of a metric is as important as the visualization of the trend of said metric over time.

[4]See **`https://github.com/prmr/JetUML`**
[5]See **`https://luckyframework.org/`**
[6]See **`http://www.lua.org/`**



October 24 2015
Commit: `d11d15c`

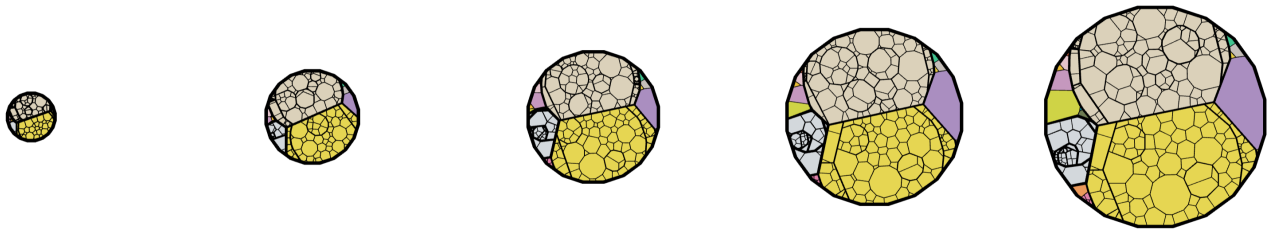
November 22 2017
Commit: `988cdef`

July 8 2019
Commit: `6a0d97e`

September 18 2019
Commit: `b382cd5`

September 8 2020
Commit: `a4f3144`

Fig. 7: Visualizing the Evolution of JetUML with `VET`



November 7 2017
Commit: `44819e6`

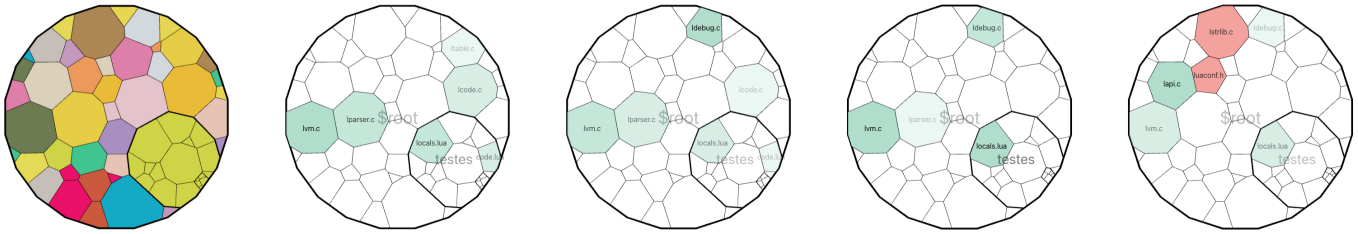
May 26 2018
Commit: `aa1cb70`

April 15 2019
Commit: `2e8781c`

March 23 2020
Commit: `097de0e`

May 30 2021
Commit: `bea9b8e`

Fig. 8: Visualizing the Evolution of Lucky with `VET`



Baseline – April 7 2021
Commit: `47cffdc`

April 10 2021
Commit: `d205f3a`

April 12 2021
Commit: `5148954`

April 16 2021
Commit: `6812971`

April 24 2021
Commit: `fc6c74f`

Fig. 9: Visualizing Last 4 Commits of Lua with the Delta Mode of `VET` (Baseline Commit Hash: `47cffdc`)

VI. Conclusions

We have presented an approach and its implementation in the form of a web-based tool called `VET`, to display evolving Voronoi treemaps. `VET` treats change as a first class entity, and thus is able to depict how systems evolve over time. `VET` is an online tool specialized on creating animated Voronoi treemaps. Figures like the ones used in this paper do a poor job at highlighting `VET`'s strength. We recommend to the interested to try out `VET`, located at **`https://vet.si.usi.ch/`**.

Acknowledgements: We thank the Swiss National Science foundation for the support through the NRP-75 and NRP-77 projects 167173 and 187353.

References

[1] M. Bruls, K. Huizing, and J. J. Van Wijk, "Squarified Treemaps," in *Data visualization 2000*. Springer, 2000, pp. 33–42.

[2] M. Balzer, O. Deussen, and C. Lewerentz, "Voronoi Treemaps for the Visualization of Software Metrics," in *Proceedings of the 2005 ACM symposium on Software visualization*, 2005, pp. 165–172.

[3] F. P. Brooks, "The Mythical Man-Month," *Datamation*, vol. 20, no. 12, pp. 44–52, 1974.

[4] F. Aurenhammer, "Power Diagrams: Properties, Algorithms and Applications," *SIAM J. Comput.*, vol. 16, no. 1, pp. 78–96, 1987. [Online]. Available: https://doi.org/10.1137/0216006

[5] A. Nocaj and U. Brandes, "Computing Voronoi Treemaps: Faster, Simpler, and Resolution-independent," *Comput. Graph. Forum*, vol. 31, no. 3, pp. 855–864, 2012. [Online]. Available: https://doi.org/10.1111/j.1467-8659.2012.03078.x

[6] Munson, John C and Elbaum, Sebastian G, "Code churn: A measure for estimating the impact of code change," in *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*. IEEE, 1998, pp. 24–31.

[7] G. Voronoï, "Nouvelles applications des paramètres continus à la théorie des formes quadratiques. premier mémoire. sur quelques propriétés des formes quadratiques positives parfaites." *Journal für die reine und angewandte Mathematik*, vol. 133, pp. 97–178, 1908.

[8] E. W. Weisstein. Voronoi Diagram. From MathWorld–A Wolfram Web Resource. [Online]. Available: https://mathworld.wolfram.com/VoronoiDiagram.html

[9] S. Lloyd, "Least Squares Quantization in PCM," *IEEE Transactions on Information Theory*, vol. 28, no. 2, pp. 129–137, 1982.

[10] B. Shneiderman, "Tree visualization with tree-maps: 2-d space-filling approach," *ACM Trans. Graph.*, vol. 11, no. 1, pp. 92–99, 1992. [Online]. Available: https://doi.org/10.1145/102377.115768

[11] M. Balzer and O. Deussen, "Voronoi treemaps," in *IEEE 2005 Symposium on Information Visualization*. IEEE CS Press, 2005, pp. 49–56.