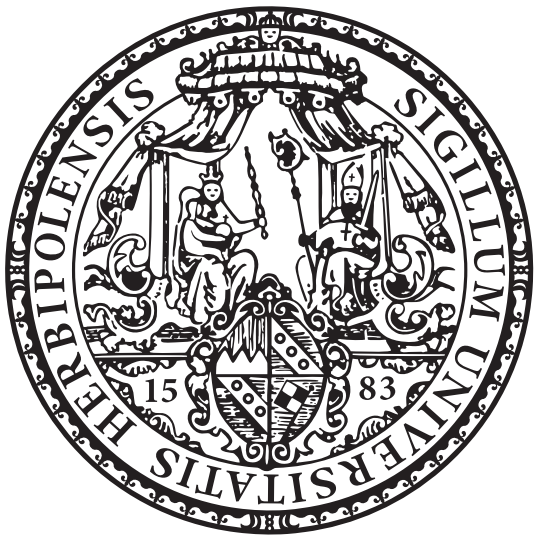
Practical Course Report

A Client-Side Implementation for the Graph Harvester

Jonas Spies

Date of Submission: 02.10.2026
Advisors: Prof. Dr. Alexander Wolff
Tim Hegemann, M. Sc.



Julius-Maximilians-Universität Würzburg
Lehrstuhl für Informatik I
Algorithmen und Komplexität

Contents

1 Introduction **4**

2 System Limitations **5**

3 Previous Work **5**
3.1 Workflow 5
3.2 Limitations 6

4 Graph Harvester as a Web Application **6**
4.1 Goals 7
4.2 Benchmarks 7
4.3 Workflow and Impact of Different Filters 8
4.3.1 Initial Vertex and Edge Selection 8
4.3.2 Additional Filters 9
4.3.3 Implied Vertices 11
4.3.4 Graph Detection 12

5 Limitations and Possible Solutions **13**

6 Conclusion **14**

Abstract

In an attempt to answer the question of what makes a graph interesting, House of Graphs have built a database of many interesting graphs. However manual insertion of new graphs being tedious creates the need for an application that automates this process. While such a system already existed, it suffered from the necessity of hosting a server due to the project being written in Python and a relatively low detection rate: Mooney, Hegemann, Wolff, Wybrow and Purchase counted roughly 40% of the extracted graphs as satisfactory. Thus, we have rebuilt a similar application in Typescript to allow client-side execution while deviating from the original workflow in some aspects in the hope of improving the detection rate. Using graph drawings from papers featured in the International Symposium on Graph Drawing and Network Visualization from 1998 to 2024, we estimated the new implementation's performance against that of the old one, and came to the conclusion that the two systems' detection rates are similar. The main reason why further improvement may be difficult is the trade-off that arises with the adjustment of some parameters: if they are changed to properly detect one instance, another instance may not be able to be detected correctly.

1 Introduction

Graph theory is a major field in mathematics and computer science due to a graph's ability to represent objects and their relations to each other in a compact and intuitive manner. This representation allows all kinds of problems to be solved in a unified approach, such as the problem of finding the shortest path from one location to another, as often applied in navigational devices.

Many of these graphs may have certain properties that make them inherently more interesting than others. For example, they may serve as a particularly illustrative example to show an algorithm's best or worst case performance, as a counterexample to a plausible conjecture, or as an extreme instance where some property, for example planarity, still holds. While it remains difficult to define what makes a graph "interesting", we use the appearance of a graph in a scientific paper as a major indicator for potential interest, leading our focus on to graphs featured in them.

In an attempt to give the word "interesting" a more concrete meaning, House of Graphs (HoG)[1] have made it their mission to create a database that contains many such graphs. However, scanning through hundreds of papers manually is not only prone to errors especially for large graphs, but also very time consuming, which is why we aimed to write a program that can automate this process.

Thus, a graph harvester app has been developed in Python to automatically scan scientific papers and heuristically extract any graph contained in them in a vector format, returning a visualization, a Graph6 String, and a link to the respective database entry or to the upload form for that graph. A user can then use this tool to upload new graphs with the click of a button.

We have rebuilt the project in Typescript to allow client-side execution of the code, thus eliminating the need of hosting a server. This opens up the possibility of integrating the harvester into the website of HoG, while achieving similar performance compared to that of the previous system. For visualization, we reused the existing frontend built with React. While following the general work pipeline of the previous implementation, we allowed ourselves to deviate from it by also adding and omitting a selected number of processing steps. For example, we widened the definition of a vertex from "a rectangle or a circle" to "any object of limited complexity that forms a closed loop". Changes like this, however, also come with new problems, for instance arrow heads being treated as vertices and such.

Using papers featured in the International Symposium on Graph Drawing and Network Visualization[2] from 1998 to 2024 to measure progress during development, we have also made some observations about the general structure of these papers and their respective graph drawings, as well as on what makes a graph drawing "difficult" to scan heuristically. Furthermore, we will highlight several trade-offs one has to consider when adjusting the system's parameters.

These observations could serve as a starting point for anybody aiming to improve

[1] https://houseofgraphs.org/
[2] https://graphdrawing.unipg.it/

the project, or as a guideline on how to make machine-readable graph drawings for researchers who wish to find their graphs included in the HoG database.

2 System Limitations

In order to scan graphs, we limit ourselves to vector graphics because those consist of multiple drawn objects that can be examined heuristically to discern between vertices and edges. While the ability to scan bitmaps as well could be promising, this would require additional effort such as working with convolutional neural networks to extract patterns. With only roughly 17% of the figures in the dataset – manually evaluated by [MHW+25] – being drawn as a bitmap, we deemed the extra effort not worth it, although future work may easily improve the performance by supporting bitmaps as well.

Besides that, we limit ourselves to undirected, simple graphs. While there may also be many interesting directed graphs or multigraphs, House of Graphs does not support them, thus we do not discern between them.

Lastly, there are many graph drawings that only include four or less vertices. While one could argue that such graphs are interesting as well, the likelihood of that instance already being on the database is high, which is why we limit ourselves to graphs that have at least five vertices, four edges, and no isolated vertices (isolated vertices will not be considered part of the graph during detection).

3 Previous Work

In this section we will give a short overview over how the previous harvester works and its limitations.

3.1 Workflow

The old implementation handles one document at a time, going through that document page by page.

Identifying Figures. As a first step on each page, the old pipeline groups drawn objects by figure, so that a matching between figure and extracted graph can be established while further narrowing the number of objects to be considered at a given time. This is achieved by a simple algorithm where each drawn object on a page starts off as its own figure. If two figures have overlapping bounding boxes, they are merged to a single figure containing the drawn objects of both. By repetition until convergence, this approach works surprisingly well to correctly identify each figure as such, which usually even gets reassigned its original figure number through another simple search. This speeds up manual evaluation because each graph produced by the harvester can that way be traced back to a unique figure in the paper.

Initial Vertex and Edge Candidates. Iterating figure by figure, the system tries to construct circles from Bézier curves, so that initially, all rectangles and circles are a vertex candidate and remaining objects are edge candidates. An initial filter clusters the vertices by size, then ensures for larger than average vertices that they do not contain any already accepted vertices and that they have some edge candidate starting inside. Any vertices that do not pass this filter will, from this point on, be treated as potential edges. After that, duplicate vertices are thrown away completely, which is important because any drawn objects with both a fill color and an outline will appear twice.

Graph Detection. From there, the harvester looks for any edge candidates starting or ending in a vertex candidate and connects pairs of segments to an edge path, if they share an endpoint and their internal angle is at least 60°. If both the start and endpoint of a segment are incident to a vertex, that segment is an *edge*. If only one endpoint is incident to a vertex, it is called a *half-orphan*. Remaining segments are *orphans*. Orphans and half-orphans are then extended in a direction to see if a vertex can be found that way and discarded if not. Finally, the system checks if any vertices lie between two endpoints and splits each such segment accordingly.

To obtain a graph from the resulting incidence matrix of vertices and incident edges, the system performs a depth-first-search to find all connected components, and returns each component as one graph. Lastly, all graphs of a single figure are wrapped together and sent to the frontend.

3.2 Limitations

According to [MHW+25], many of the graphs that could not be extracted by the previous pipeline could at least be explained due to one of the following patterns:

1. Node-link diagrams that are not a graph often cause false positives.
2. Two-in-one drawings, as well as drawings with an underlying grid, usually produce a graph with too many edges.
3. Disconnected components are recognized as separate graph drawings.
4. The pipeline fails to recognize vertices that are huge, heavily-annotated, text-only, unusually shaped, isolated, implicit, too bright, or ambiguously placed.
5. Edges are ignored if they are too far from their target, implicit, too bright, feature tight angles, or connect to multiple vertices.

4 Graph Harvester as a Web Application

While the general workflow is similar to that of the previous graph harvester, we will now discuss differences between the two systems as well as resulting problems and how they are solved in this implementation.

4.1 Goals

The goal is a graph harvester that runs client-side in a browser – potentially integrated into House of Graph's website – so that the user can upload PDF files and expect to reliably be shown a figure in the paper, as well as the corresponding graph. The user can then decide if they want to edit the graph using HoG's built-in editor, or if they want to upload the graph to the database. Because the user is ultimately in control, it is best to have as few false negatives as possible, even if that means having more false positives.

Additionally, we have tried implementing features to improve on some of the previous system's shortcomings, such as the detection of unusually shaped or implicit vertices, and edges bending in a tight angle.

4.2 Benchmarks

Besides the web application, we have built a second entryway to the pipeline which runs through two hardcoded directories, one containing a set of reference graphs, and one containing a set of scientific papers in PDF format. The system will iterate the pipeline over all papers and in each step try to match as many of its graphs to an equivalent reference graph belonging to that paper. While the equivalence check is deliberately simple, only comparing the number of edges and vertices, we have found it sufficient for the purposes of this benchmark due to the inherently small number of graphs featured in a single file. This allowed us to focus on improving the system rather than implementing a more intricate algorithm for graph isomorphism.

The reference graphs come from the old pipeline, but were manually checked to only contain graphs graded 'A', a rating given when the extracted graph is isomorphically identical to the depicted graph, i.e. an instance where the previous system succeeded without any errors. This means that any time 'recall' is mentioned, it is by no means a measure against ground truth, but more of a measure of agreement with the old implementation, given instances the old implementation performed well at.

We initially attempted to get closer to ground truth by manually constructing further reference graphs for instances that the previous implementation failed to detect accurately. However, this quickly turned out to be very time consuming and error prone, as even small graphs could easily take us 15-20 minutes to reconstruct due to tasks such as mapping vertices from the drawing to an adjacency list. This led us to the decision to throw away the few extra instances we created and instead focus only on the graphs originally graded as 'A'.

With only 248 out of the total 706 graphs in the data set being graded 'A', there is a lot of potential improvement our benchmark is unable to show, which means that generally, our benchmark underestimates the performance of the new implementation. For example, a graph harvester that could accurately determine all the 458 graph drawings the old application failed at, but none of the 248 graph drawings the old application succeeded at, would still achieve 0% recall in this benchmark.

With that being said, the most recent benchmark over all papers that included at



(a) Taken from [AOV24]. (b) Taken from [LMT18].

Fig. 1: Instances from the dataset where the new implementation outperforms the old one.

least one graph graded 'A' was able to match 222 out of 248 graphs for a recall of 89.5% and found a total of 578 graphs for a precision of 38%. Note that, even though precision is fairly low, the actual precision against ground truth is likely higher than what this benchmark can indicate due to the same reasons discussed before. We have additionally manually found some instances where our implementation performs better than the old one – examples shown in Figure 1 – which leads to our conclusion: Even though the new implementation fails at some instances the old implementation succeeded at, it also succeeds at some instances the old implementation failed at, so that the two systems seem overall similar in performance.

4.3 Workflow and Impact of Different Filters

Similar to the old graph harvester, we use MuPDF[3] to extract vector graphics page by page, then group them together into figures by overlapping bounding boxes. From there, we use various heuristics to interpret these vector drawings as a graph drawing.

4.3.1 Initial Vertex and Edge Selection

The first difference to the old system arises at the initial selection of vertex and edge candidates: We iterate over every drawn object and simply check if it forms a closed loop. With a lot of papers drawing letters as a vector graphic, this produced many instances where letters would be interpreted as vertices, which is why we eventually also added a complexity filter, limiting how many Bézier curves are allowed in a vertex. Additionally, many drawings feature directed edges or arrows to highlight certain vertices. The arrow heads would then be falsely interpreted as a vertex, so that we built a simple arrow head detection that would mark any object with a sharp angle of 45° or less as an arrow and thus reject it as a vertex. Any object not selected as a vertex would then be broken down into its separate strokes, interpreting each of these strokes as an edge candidate. Rejected arrows are currently reused as edge candidates because the current detection is too broad, so that throwing them away would cause problems for many graph drawings. A potential solution for this is briefly discussed in Section 5.

[3] https://mupdf.readthedocs.io/en/1.28.2/

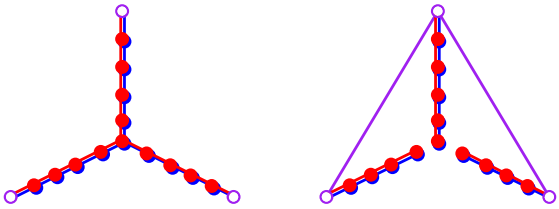


Fig. 2: Taken from [HKvK+13].

4.3.2 Additional Filters

While the selection of vertex candidates was deliberately lax initially, it became clear that further filtering the selection of vertex candidates was key to improving performance: an earlier version without any of the following filters only found 30 'A' graded graphs, as opposed to the current version which found 222. One filter that misses from the old system, however, is that of checking for duplicate vertices, i.e. filtering one of two vertices if both have the same center. While there are many other filters in place that handle duplicate vertices, the absence of this filter is actually a prime candidate for the explanation as to why sometimes vertices seemingly blow up in size compared to the original drawing, as discussed in Section 5.

Anybody who wants to experiment with these filters themselves can find relevant parameters in the file "graph_detection.ts", with some additional parameters in the file "geometry_utils.ts".

Merging Overlapping Vertices. Due to the nature of how vector graphics are drawn, most vertices actually produce two objects: one for the outline and one for the interior color. If both survive the initial selection process, it becomes essentially random which of the two objects is selected to become linked to an edge, due to an edge only being able to store one vertex per endpoint. This would cause problems during the last step discussed in Section 4.3.4, so that many vertices would be split into two and some components of a graph may end up missing entirely. As a result, an initial filter which merges any two vertices into a large single one, if they overlap by a certain threshold, became the most important feature to be implemented, boosting the detection rate from an initial 30 instances, all the way up to 173 instances found. However, the following filter would largely take over this role, so that actually merging vertices started becoming more of a backup system in our implementation. Still, instances like the one shown in Figure 2, where there are two offset circles depicting the same vertex, prove that a feature like this remains necessary.

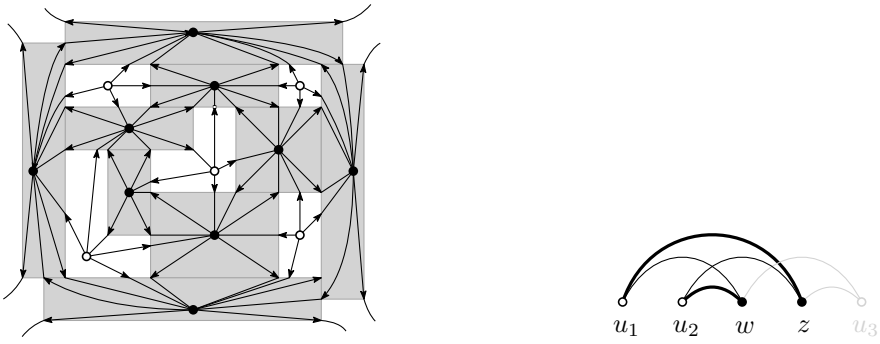
Clustering Vertices by Area. Largely copied from the old graph harvester, this filter first clusters vertices by the size of the area they take up, then splits each cluster into two groups: the accepted group, and the group up for further examination. If a cluster contains too few elements, or if its mean area is larger than the average over the remaining clusters, it would be sent to further examination, while other clusters would be accepted.

A cluster up for further examination then needs to ensure that it does not contain any center point from the set of already accepted vertices. This check largely ensures that if there is one object for the outline and one for the interior of a vertex, just one, usually the interior due to its smaller area, is chosen. However, the parameter of a minimum cluster size interestingly enough turned out to be a hugely influential parameter: While we eventually got it down to a value of 3, pushing it down further comes at a huge cost of detection rate, likely due to less actual vertices being accepted without any further examination. This creates the opportunity for huge objects to get accepted as a vertex, if they are examined before any of the actual vertices that it overlaps. That way, the huge vertex would remain accepted, and later merge with all actual vertices into one big vertex, swallowing parts of the actual graph drawing. While we believe that there could be better ways to balance an initial selection of vertices, this one proved good enough, so that we found it hard to manually find an instance where this approach causes problems.

Height-to-Width Ratio. Another filter that seemed promising at first was that of a filter based on the height to width ratio of a vertex. We eventually optimized the parameter so that any vertex that has less than 75% of its width in height, or of its height in width, would be rejected. While this initially gave a small boost in recall, recent tests have shown no difference. A further optimization could be to only use the filter, if some vertices still remain after, but in its current form, the impact of the filter seems to be weak.

Size of Vertex Compared to Total Size of Figure. While simply filtering vertices if their area is too large relative to that of the figure seems like a good idea, in practice other filters already seem to do this job. As a result, no difference in recall could be observed when changing the threshold for this variable within reasonable bounds. For example, filtering any vertex that takes up 20% or more of the total area, compared to already filtering it if it takes up 5% of the total area, does not change recall at all, and setting the threshold too low naturally hurts recall, rather than helping it. However, one interesting observation is that contrary to intuition, the total number of detected graphs is higher with a stricter threshold: having detected a total of 611 graphs with a threshold of 5% as opposed to just 578 graphs detected with 20%. This could indicate that there are some niche instances the old pipeline could not properly detect, that do rely on this filter, which motivated us to keep the threshold relatively low at 5%.

Brightness. Filtering objects, not just vertices, by brightness could be a powerful tool to get rid of many objects that make an instance difficult to detect, such as grids and background colors. For instance, Figure 3a shows a graph drawing with a light gray grid laying beneath. Creating a brightness scale from 0 (black) to 1 (white) and completely throwing away any object over a certain threshold can make many instances easier. For example, a threshold of 0.87 would cause the aforementioned graph to lose its underlying grid. However, overall 0.87 seems like it is already too strict, as many instances such as the one shown in Figure 3b break at this point as well. In total, a threshold of 1



(a) Taken from [KNU15]. (b) Taken from [BGMN20].

Fig. 3: Two example graph drawings illustrating the dilemma of choosing a brightness threshold.

compared to one of 0.87 detects 222 as opposed to just 215 graphs from the data set, which unfortunately indicates that, while there is a trade-off of sacrificing some graphs to be able to detect others, it is generally not worth taking currently since it means losing graphs in total.

Arrows. While the old implementation with its strict definition of a vertex had no problem with arrow heads, our new implementation does suffer from detecting any arrow head as a vertex, leading to many falsely detected instances. To fix that, we implemented a simple arrow head detection, that would reject any object as vertex candidate, if it contains a sharp angle, namely 45° or less. As a small optimization, the system actually does not compute the internal angles, but instead the cosines, where a cosine of 0.707 or larger corresponds to such an angle.

In order to compute the internal cosines of an object, we simply do pairwise comparisons between any two strokes that meet in a common point. With the first iteration of this algorithm simply walking along the outline of an object to find these pairs, we quickly encountered a bug that caused many arrow heads to not be detected. The fix lied in the interesting observation, that some objects are actually drawn in a way where they close the loop through walking along the perimeter, and then close the loop again by connecting the first and last endpoint of the object's path. Since those are already identical, this resulted in many arrow heads having one line segment of length 0, so that if two adjacent line segments form a tight angle, that pair would never be checked because along its perimeter, a line segment of lenght 0 lies between the two segments.

By fixing that bug, arrow head detection allowed correct detection of twelve additional instances, so that instead of 210 graphs, 222 could be found.

4.3.3 Implied Vertices

A large difference to the old pipeline is an attempt to detect implied vertices. To this end, we simply check for an instance if there are any vertex candidates left after initial filtering. If not, the system will look for any point where three or more edges meet in

an endpoint, and create a vertex incident to these edges at that position. Eventually we tried creating implied vertices at a point even if just two edges meet there, but this caused hundreds of false positives due to any drawn object with at least six segments, such as letters, suddenly turning up as a detected graph, which led us to revert this decision. However, none of the graphs with implied vertices are represented in the benchmark due to the old implementation not having this feature. Determining the actual impact of the feature on the detection rate thus necessitates further manual evaluation.

4.3.4 Graph Detection

With the set of vertices being determined, the next step is checking which edges are incident to that vertex. Although most vertices lie on an endpoint of an edge segment, there are many instances where vertices lie between two endpoints, which led us to the following algorithm:

First, divide the segment into many sample points (our results come from using 30 points), with the first and last being the two endpoints and the others being equally spaced along the segment. Secondly, check for each vertex if any sample points lie within its bounding box, scaled by some lenience factor (best results with 1.07). The concrete implementation of the range query is through RBush[4]. The result is an incidence list of edge segments for every vertex that will afterwards be used to split every segment so that no segment has a vertex lying between its two endpoints, and every edge segment knows if it is incident to a vertex at each of its endpoints. With an edge being a path from one vertex to another, there may be instances where multiple vertices are incident to a single endpoint. In such a case, we take the vertex minimizing the distance to the endpoint, which is a solution that seemingly has not caused any problems.

After that, we use KDBush[5] to find all orphaned edge endpoints incident to another orphaned edge endpoint, i.e. a pair of incident endpoints that are both not incident to any vertex. Note that while the previous system only links edge segments if they do not form a tight angle (60° or less), we have not implemented this check. While the search radius has caused some troubles, the best results could be achieved with the search radius being either the stroke width, or 0.25px, depending on which is larger, with very short segments not getting the guarantee of at least 0.25px. This solution seems able to handle two specific edge cases that were particularly prevalent: One occurs in instances where two strokes form one edge, yet they don't exactly meet in a given point, i.e. there is an offset between two consecutive edge segments (see Figure 4a). The other occurs whenever two separate edges come very close to each other at a given endpoint, which might cause the wrong edge segments to be linked to each other (see Figure 4b).

This allows chaining different segments together into one large edge, which is important for many instances: in an earlier version where this was not possible, only 22 out of the 248 'A' graded graphs could be matched. After implementing the chaining of edge segments, the recall jumped up by a relative 36%, up to a total of 30 matched instances.

[4]https://github.com/mourner/rbush
[5]https://github.com/mourner/kdbush



(a) The two orange segments should be linked, yet a search radius that is too small will not connect the two segments into one edge.

(b) The purple segments should be linked, yet a too large search radius might cause one of them to connect to the orange clutter instead.

Fig. 4: Two examples that highlight why choosing a search radius can be tricky.

Lastly, remaining half orphans are rescued by linearly extending them into the direction they point at. For Bézier curves, the extension goes into the direction obtained by shooting from the last control point towards the orphaned endpoint. Since this implementation also caused many false positives, we had to carefully limit the extension, up to a maximum of 30 pixels, or seventeen percent of the edge segment's length. During tuning of these parameters, it turned out that the relative extension threshold is much more limiting than the absolute one: Although there were two instances that could only be detected with at least three pixels of absolute extension, and one that needed nine pixels, any increase beyond that neither positively nor negatively influenced the detection rate.

The resulting incidence list can then be turned into a graph using depth-first-search, just like the old graph harvester did.

5 Limitations and Possible Solutions

Arrow Head Detection. The system currently relies on reusing arrow heads as edge segments. This causes many perfectly fine edges to attach the arrow's stroke segments to its end, rather than the vertex it is incident to. While that does not cause any problems in itself, it causes the last segment of the edge to be very short. That way during the edge extension process for half-orphans, the relative edge extension threshold causes many segments to be extended much less than they could be. Thus getting rid of arrows being used as edge segments could be the key to detecting more graphs accurately.

However, in order to enable that, it would be interesting to analyze cases where an arrow is actually important for edge detection. Possible indicators could be size of the arrow, and number of vertices it is incident to. For example, if the arrow consists of three line segments, there could be three vertices the arrow is incident to. If there are more or less vertices in its vicinity, the arrow might be safe to discard as an edge.

Large Vertices. The implementation often bloats vertices up to be much larger than they appear to be in the original drawing, causing problems by finding additional incident edges and falsely merging two bloated vertices into one. It remains to be seen if this is due to a bug, or some other phenomenon that can be counteracted. A plausible explanation could be, that some papers might draw their vertices with a blown up interior that they then clip to the outline of the vertice. Since clipping paths are not considered, that blown up interior stays as large, and might be able to sneak through the filters designed to catch large vertices.

6 Conclusion

We built a Graph Harvester with a performance comparable to that of the previous system. A test of the previous implementation conducted in [MHW+25] found that roughly 40% of the extracted graphs were satisfactory. Our benchmark focused on 248 out of the 706 extracted graphs that were deemed without errors and was able to reproduce 222 of them. Since this benchmark only covers graphs that the old Graph Harvester handled correctly, we additionally identified a few examples in which the new implementation outperformed the old one.

Despite deviating from the previous implementation in some aspects, the similar performance of the two systems suggests that purely heuristic approaches may face inherent limitations: For example, a threshold that correctly determines whether two endpoints are close enough to be considered adjacent for one drawing may lead to an incorrect interpretation of another. With different drawing styles requiring different parameters, a substantially better detection rate may necessitate a different approach.

Despite that, there is still room for improvement for our Graph Harvester. Particularly some of the features missing from the old system, or a refined implementation of the system's weaker aspects such as its arrow head detection could be added.

The Typescript implementation also facilitates deployment as a web application, which may make the Graph Harvester more accessible to users than the previous system.

References

[AOV24] Oswin Aichholzer, Joachim Orthaber, and Birgit Vogtenhuber: Separable drawings: Extendability and crossing-free hamiltonian cycles. In Stefan Felsner and Karsten Klein (editors): *32nd International Symposium on Graph Drawing and Network Visualization, GD 2024, Vienna, Austria, September 18-20, 2024*, volume 320 of *LIPIcs*, pages 34:1–34:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2024, 10.4230/LIPICS.GD.2024.34.

[BGMN20] Sujoy Bhore, Robert Ganian, Fabrizio Montecchiani, and Martin Nöllenburg: Parameterized algorithms for queue layouts. In David Auber and Pavel Valtr (editors): *Graph Drawing and Network Visualization - 28th International Symposium, GD 2020, Vancouver, BC, Canada, September 16-18, 2020, Revised Selected Papers*, volume 12590 of *Lecture Notes in Computer Science*, pages 40–54. Springer, 2020, 10.1007/978-3-030-68766-3_4.

[HKvK+13] Ferran Hurtado, Matias Korman, Marc J. van Kreveld, Maarten Löffler, Vera Sacristán Adinolfi, Rodrigo I. Silveira, and Bettina Speckmann: Colored spanning graphs for set visualization. In Stephen K. Wismath and Alexander Wolff (editors): *Graph Drawing - 21st International Symposium, GD 2013, Bordeaux, France, September 23-25, 2013, Revised Selected Papers*, volume 8242 of *Lecture Notes in Computer Science*, pages 280–291. Springer, 2013, 10.1007/978-3-319-03841-4_25.

[KNU15] Jonathan Klawitter, Martin Nöllenburg, and Torsten Ueckerdt: Combinatorial properties of triangle-free rectangle arrangements and the squarability problem. In Emilio Di Giacomo and Anna Lubiw (editors): *Graph Drawing and Network Visualization - 23rd International Symposium, GD 2015, Los Angeles, CA, USA, September 24-26, 2015, Revised Selected Papers*, volume 9411 of *Lecture Notes in Computer Science*, pages 231–244. Springer, 2015, 10.1007/978-3-319-27261-0_20.

[LMT18] Giuseppe Liotta, Fabrizio Montecchiani, and Alessandra Tappini: Orthopolygon visibility representations of 3-connected 1-plane graphs. In Therese Biedl and Andreas Kerren (editors): *Graph Drawing and Network Visualization - 26th International Symposium, GD 2018, Barcelona, Spain, September 26-28, 2018, Proceedings*, volume 11282 of *Lecture Notes in Computer Science*, pages 524–537. Springer, 2018, 10.1007/978-3-030-04414-5_37.

[MHW+25] Gavin J. Mooney, Tim Hegemann, Alexander Wolff, Michael Wybrow, and Helen C. Purchase: Universal Quality Metrics for Graph Drawings: Which Graphs Excite Us Most? In Vida Dujmović and Fabrizio Montecchiani (editors): *33rd International Symposium on Graph Drawing and Network Visualization (GD 2025)*, volume 357 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 30:1–30:20, Dagstuhl, Germany, 2025. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 10.4230/LIPIcs.GD.2025.30.