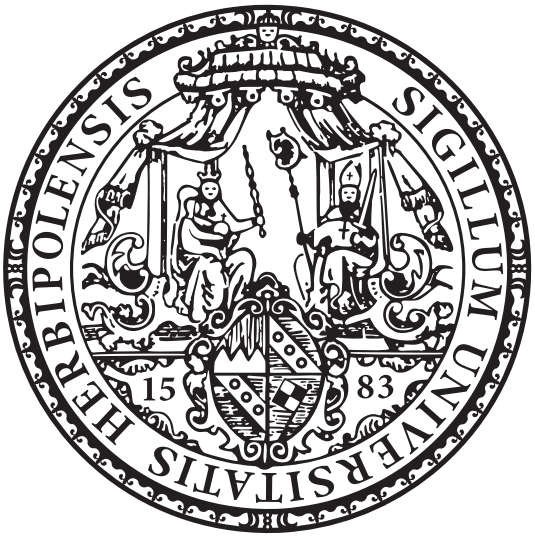
**Practical Course Report**

# Graph Harvester

Julius Deynet

| | |
|---|---|
| Date of Submission: | January 21, 2025 |
| Advisors: | Tim Hegemann |
| | Sebastian Kempf |
| | Prof. Dr. Alexander Wolff |




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

# 1 Introduction

In the field of graph theory, some graphs play a crucial role in many publications. Therefore, it is helpful to have a collection of interesting graphs that have been used in papers before at hand. For this, the website houseofgraphs.org [1] is a useful tool. Originally published in 2012, one can search, draw, and upload graphs with the explicit goal not to collect all possible graphs, but only interesting ones. Graphs can be searched by many criteria, from basic parameters like the number of edges or vertices to more advanced metrics like the clique number. This collection of graphs, however, is far from being complete. From 293 potentially interesting graphs (with at least seven vertices and maximum vertex degree of at least three) appearing in figures from papers presented at GD 2023, only 57 were already contained in the HoG database.

To solve this problem, we present Graph Harvester, a website that extracts graphs from drawings of graphs in PDF files. Whereas there has been work on extracting graphs from bitmap images [2], we focus on the simpler problem of extracting graphs from vector data which is the main format used in publications nowadays. Graph Harvester works as follows. The user uploads any publication as a PDF file. Then the website shows, for each figure in the file that contains a drawing of an interesting graph, the figure together with a drawing of the extracted graph for review. For each extracted graph, the website computes a representation in the formats accepted by HoG and checks whether the graph is already present in the HoG database. Graph Harvester can be found at `go.uniwue.de/graph-harvester`.

In the following, after presenting some related work, a more detailed explanation on how the software works will be provided. For that, an example is used, which also showcases the final result of the website.

# 2 Related Work

First, some related work is discussed. The similar approach of recognizing graphs from images is described first, followed by a description of the House of Graphs website and the graph6 string encoding for graphs.

## 2.1 Graph Recognition from Images

While we do not know of any work on detecting graphs from vector data, Auer *et al.* addressed a similar task of detecting graphs from images [2]. Here, the goal was to reconstruct graphs from hand drawings, assisting the workflow of sketching graphs by hand and using them in graph drawing tools. This is achieved through the following steps: First, each pixel is classified into three classes: background, edges, and vertices. This is done by first binarizing the pixels of the image to extract all object pixels and then eroding the regions of the latter. As long as edges are drawn with a width significantly smaller than vertices and vertices are filled, edges can be eroded away, allowing the

extraction of only vertex pixels. Next, the now ternary image is skeletonized to discard duplicate information and traversing edges. At intersections, the direction is used to determine where an edge continues. Being able to handle intersections is a crucial contribution of this work. Lastly, the positions of all detected vertices are stored and bends assigned to each edge.

This approach worked somewhat well, correctly identifying five out of ten provided hand drawn tests. When tested on the Rome graphs drawn using Gravisto, OGR recognized 93.24% correctly. However, skeletonization makes it prone to errors if many intersections occur in a small area, and detecting vertices via erosion does not allow for unfilled vertex drawings [2].

## 2.2 House of Graphs

The House of Graphs website (HoG) is an online repository dedicated to cataloging interesting graphs for research in graph theory and related fields. Originally introduced by Brinkmann et al. [1] in 2013, the platform provides a searchable and extensible database of graphs with various properties that serve as a valuable resource for researchers investigating graph-based problems including counterexamples, extremal structures, or small graphs with specific characteristics.

In their initial paper Brinkmann et al. [1] outline the motivation for the website and its key functionalities. The authors highlight the importance of providing an easily accessible platform where researchers can both contribute new graphs and search for existing ones based on various structural properties. The search capabilities of HoG allow users to filter graphs by parameters such as number of vertices, edges, degree sequence, or predefined invariants (e.g., girth, chromatic number, or diameter) which can significantly reduce the time and effort needed for exploring particular graph families or finding relevant examples. The platform also includes a "Graph of the Day" feature, designed to continuously showcase graphs with intriguing properties to stimulate research ideas.

In 2023, the website was completely rebuilt to bring the underlying technologies up-to-date. The changes were described in the paper "House of Graphs 2.0: A database of interesting graphs and more" by Coolsaet et al. [3]. Furthermore, additional features such as a tool for drawing and editing graphs were introduced and described.

Due to its reliance on user contributions and the absence of a fixed definition for "interesting graphs", the website offers a collaborative environment. HoG's contribution to the community has been further solidified by its role in enabling researchers to efficiently share and access difficult-to-find graph instances.

### 2.3 Graph6 Format

The graph6 format is widely adopted for encoding undirected graphs in a compact, text-based way. Originally developed for use in graph theory research, particularly within the domain of combinatorics, graph6 provides an efficient way to store and share graph data. A key feature of the graph6 format is its ability to represent graphs with up to $2,147,483,647$ vertices, making it suitable for a broad range of applications involving large-scale graphs.

Each graph6 string begins with a header that encodes the number of vertices, followed by a binary representation of the adjacency matrix of the graph. The adjacency matrix is flattened and stored as a sequence of bits, which are then grouped into 6-bit chunks and converted to printable ASCII characters. This approach results in a highly compressed, yet human-readable string that can be easily handled by computers and shared through text-based communication [4].

Graph6 has become a standard in graph theory research, largely due to its inclusion in tools like *Nauty* [5] and *Traces* [6], which are widely used for graph isomorphism testing and automorphism detection. Additionally, the format is supported by various graph-related software libraries, such as *NetworkX* [7] and *SageMath* [8]. In summary, the graph6 format is a robust, efficient, and widely supported method for encoding and sharing graph data, which has contributed to its adoption in various research and practical applications within the field of graph theory.

## 3 Approach and Implementation

Here, the pipeline for extracting graphs and their drawings from publications in PDF format will be explained in more detail. The most notable steps are the extraction of geometric shapes and the detection of vertices and edges. Lastly, the GUI on the website that displays the extracted results will be presented and explained.

### 3.1 Backend

When trying to recreate a graph embedded in a PDF, one has to detect the vertices and edges forming the graph. This is not trivial as the PDF format has no internal representations of these but only of geometric objects like cubic Béziers curves, rectangles or lines. Not even circles (i.e., potential vertices) are stored directly as they are not supported by the format but commonly represented using four cubic Béziers curves. Therefore, the following geometric objects need to be extracted first:

- Circles and squares that represent potential vertices and
- straight lines and Béziers curves that represent potential edges.

After that, we try to connect any potential vertex to any other potential vertex using any combination of lines and curves. Only if this is successful, we categorize the two

circles/squares as vertices and the curve connecting them as an edge. The required steps are explained in more detail in the following.

### 3.1.1 Extracting Geometric Objects from PDF

First, the individual figures and their components are extracted from the given PDF. The document is split into textual and graphical elements using PyMuPDF, a Python library for data extraction, analysis, conversion, and manipulation of PDF documents. These graphical components are then clustered to find cohesive groups that contain figures in vector format. The corresponding captions are found using a keyword-based approach. This step is based on and similar to the KIETA pipeline [9]. Last, a JSON object is created containing the detected figures, their objects, an image representation of the figure and, optionally, a found caption (if it exists).

### 3.1.2 Processing the Detected Geometric Objects

When processing the resulting JSON object, there are three main geometric objects of interest: lines, cubic Bézier curves, and rectangles. These are simply read from the JSON output of the above-mentioned KIETA pipeline.

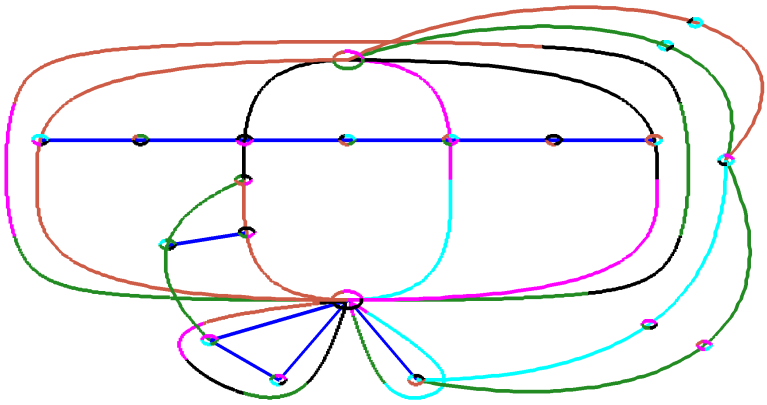


**Fig. 1:** Example of a graph drawing for Figure 8(b) of a paper of Fink et al. [10] split up into its graphical components consisting of rectangles, line segments and Bézier curves. Bézier curves are colored randomly in our drawing.

### 3.1.3 Detecting Circles from Bézier Curves

As a first step, after extracting all basic geometric shapes from the PDF, we identify cubic Bézier curves that represent a circle. A possible representation of circles can be seen in Figure 2.

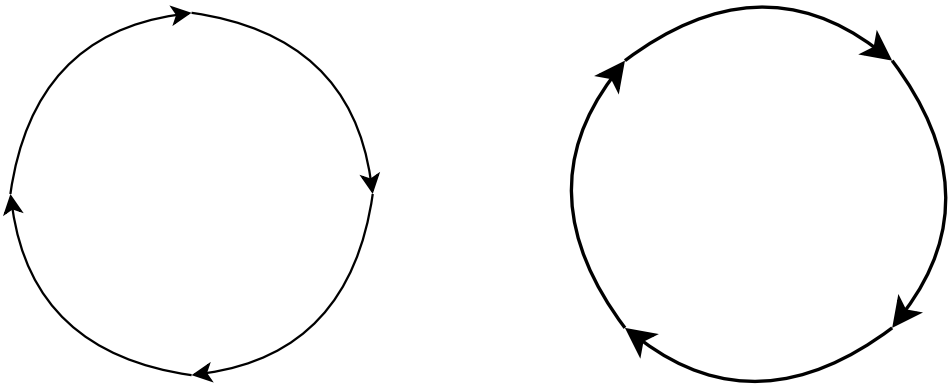


**Fig. 2:** Example of how circles in the form of four cubic Béziers curves could be found in a PDF.

Detecting circles is important as they form possible vertices. For curves to be converted to a circle, the following conditions must apply:

- The curves must form a circle, meaning the ending point of one curve must be the starting point of the next.
- At least three curves must form the circle.
- We determine the center of the circle by averaging over all curve starting points. The distance of any starting point to the center must not differ from the average distance (i.e., radius) by more than a specified threshold. This hyperparameter is called *CIRCLE_THRESHOLD*, specifying the percentage of the radius a distance may differ. This is done to filter out oval-shaped objects, which typically do not represent vertices.

These criteria allow for circle detection independently of the rotation of the curves forming the circle or the number of curves used. Curves forming a circle are typically stored consecutively, which we require for correct identification.

The results of this step are visualized in Figure 3.

### 3.1.4 Vertex Filtering

In PDF format, shapes such as rectangles or circles are often drawn twice due to different drawing styles. Additionally, these shapes are frequently used for highlighting purposes. Therefore, filtering is necessary to ensure only plausible vertex candidates remain.

To minimize false positives, circles are first filtered based on their size. A parameter, *KEEP_VERTEX_THRESHOLD*, is introduced to define the allowable percentage by which vertex candidates can differ in area and still be considered the same size. The following steps are then performed:

1. Vertex candidates are grouped by area.

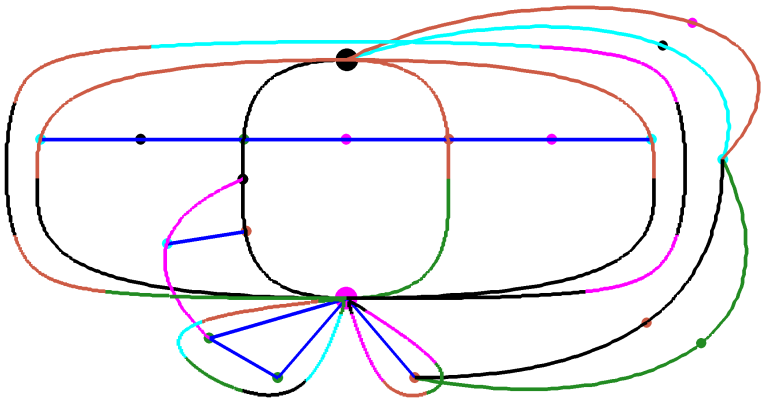


**Fig. 3:** The drawing of Figure 1 after circle detection. Circles or rectangles that contain or have an atypical area in comparison to the other vertex candidates are discarded, as well as candidates that overlap. Bézier curves and vertex candidates are colored randomly.

2. Groups with four or fewer entries are temporarily discarded since vertices in graph drawings are typically drawn with similar sizes.
3. For the remaining groups, we check if the average area of any group is three times larger than the others. If so, we check if these larger entries contain other vertex candidates. If they do, they are discarded. This step filters out highlighting in the original drawing that may appear multiple times and is not eliminated in the previous step.
4. The vertex candidates previously discarded (from Step 2) are re-evaluated. If a candidate contains no other vertex candidates and has at least three edge candidates (such as line segments or Bézier curves) originating from it, it is reinstated. This accounts for cases where certain significant vertices are drawn larger than usual. An example for this is the graph drawing used here for illustration, e.g., the big black and the big pink vertex in Figure 3.
5. Finally, the (implicit) components of the discarded vertex candidates (i.e., Bézier curves for circles and line segments for rectangles) are recovered, as they represent valid edge candidates. An example for the result of this step can be found in Appendix A.

Overlapping vertex candidates are removed, as they are not part of a valid graph and mostly a product of drawing styles of the containing PDF file. This is done by filtering out vertices contained in other vertices first, and overlapping vertices second. In the latter case, the vertex with the larger area is preserved.

Now that all potential vertices are found, the next step is to find all curves representing an edge, i.e., connecting two vertices.

### 3.1.5 Edge Detection from Line Segments

To detect edges from line segments, the basic task is to find all line segments that connect vertex candidates. This is done by iterating over all line segments, and for each line segment we check if its starting and ending points lie on two vertex candidates. Here, to provide an error margin in the drawing, again the hyperparameter *CIRCLE_THRESHOLD* defines by how many percent outside a vertex candidate's radius/width/height the end of a line segment/curve may lie to still be considered as connected. If this is the case, the line segment represents an edge that connects the two vertices. If not, we try to extend the existing line segment using any other line segment that is connected to the point or the points that are not connected to a vertex candidate. This is done until two vertex candidates are connected to each other or no further line segment is connected to the polygonal path. Only in the former case, the path is interpreted as an edge and the two vertex candidates as vertices.

After an edge has been found, it is checked if there are further vertex candidates lying on the line segment or polygonal path forming the edge. If this is the case, the additional vertex candidates are seen as vertices and connected to their two neighbors on the path.

Line segments and paths that do not start and end on a vertex candidate are not considered, as they mostly represent axes etc. in graph drawings.

### 3.1.6 Edge Detection from Curves

To detect edges from Béziers curves, the basic task is to find all Béziers curves that connect vertex candidates. This is done by iterating over all Béziers curves. For each curve, we check if its beginning and ending points lie inside two vertex candidates. Here, to provide an error margin in the drawing, again the hyperparameter *CIRCLE_THRESHOLD* defines how many percent outside of a vertex candidate's radius/width/height the end of a line segment/curve may lie to still be considered as connected. If this is the case for both the start and end point, the curve represents an edge connecting the two vertices. If not, we try to extend the existing curve using any other curve or line segment that is connected to the point or points that do not lie inside vertex candidates. This is done until two vertex candidates are connected to each other or no further curve or line segment is connected to the path of consecutive Béziers curves. Only in the former case, the path is interpreted as an edge and the two vertex candidates as vertices.

After an edge has been found, we check if there are further vertex candidates lying on the curves or lines forming the edge. If this is the case, the additional vertex candidates are seen as vertices and connected to their two neighbors on the path.

After these two steps, all vertices and edges have been extracted from the PDF/JSON and its geometric objects.

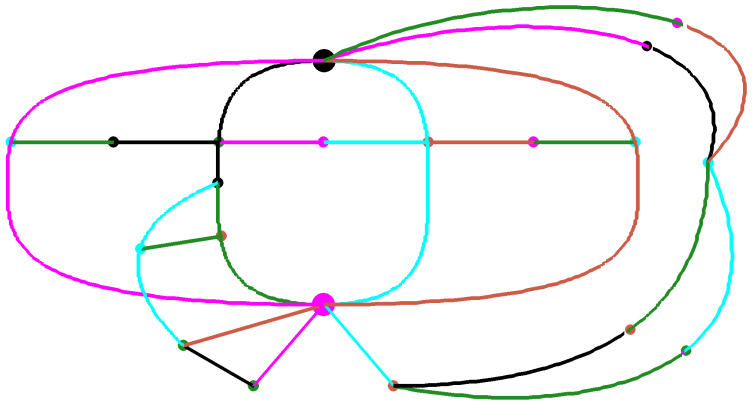


**Fig. 4:** The drawing of Figure 3 after edge detection. All paths of line segments or Bézier curves that connect previously prefiltered vertex candidates (i.e., circles or rectangles) are detected as edges. Vertices and edges are colored randomly.

### 3.1.7 Conversion to Adjacency Matrix

Next, the lists of vertices and edges have to be converted to a format interpretable by House of Graphs (HoG). This is done by creating an adjacency matrix of the extracted graph. Since a figure may contain multiple or disconnected graphs, we split the extracted graph into its connected components if needed. The result is a list of adjacency matrices, where each entry represents a connected graph.

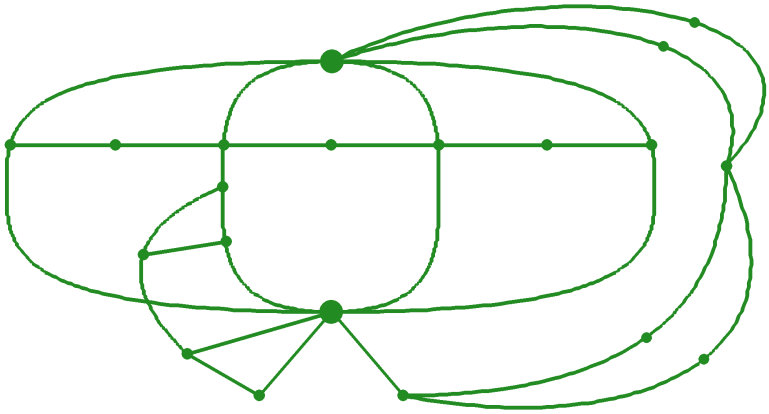


**Fig. 5:** The drawing of Figure 4 after splitting it into connected graphs. These are converted to their graph6 string representations separately. Drawing is colored by connected components.

### 3.1.8 Final Filtering

As a last step of filtering, graphs with less than seven vertices and maximum vertex degree of less than three are discarded. This can be explained by the fact that without meeting these two metrics, the graph is likely not interesting or already contained in HoG. The requirement for the vertex degree furthermore helps to prevent false positives such as function plots from being detected as graphs.

### 3.1.9 graph6 Conversion and HoG Integration

Lastly, the remaining interesting graphs are converted from their adjacency matrix to graph6 representation. This representation is then used to make an API call to HoG. The graph is then internally converted to its canonical form by the API and checked if it already exists on the website. If this is the case, among others, an identifier is returned [3].

### 3.1.10 Providing Information to the Frontend

Now that all graphs present in the provided PDF have been extracted, this information has to be sent to the frontend. For this, a JSON is created. It contains the graph6 strings and HoG IDs, as well as the graphical information in the form of all circles, lines, rectangles, and Béziers curves that have been found. Furthermore, the bounding box of the original figure extracted from the PDF is provided. To enable the user to identify the displayed graph more easily, the caption of the figure is also included (assuming one was found). Finally, to further ease the identification and to make Graph Harvester's result verifiable, a reference image of the figure directly extracted from the PDF is added.

Now, all relevant information has been extracted from the PDF and has been provided to the frontend. Next, the frontend displays this information.

## 3.2 Frontend

The frontend consists of a basic website written in JavaScript. Here, the user can upload a PDF file, which is sent to the pipeline described in Section 3.1.1. After the backend has extracted the graphs contained in the uploaded file and replied with the information described above, the frontend composes an SVG image per figure based on the geometric components. This reconstruction is then displayed next to an image of the original figure. The frontend also lists the graph6 strings for all graphs found in the figure. An example output can be found in Figure 6.

If the extracted graph is already contained on the HoG website, a link to the entry is provided. If not, a link is generated where users can directly add the extracted graph while preserving its drawing. This is done by embedding the graph6 encoding, as well as the positions of the vertices (in the order that was used to create the graph6 string) into the link.

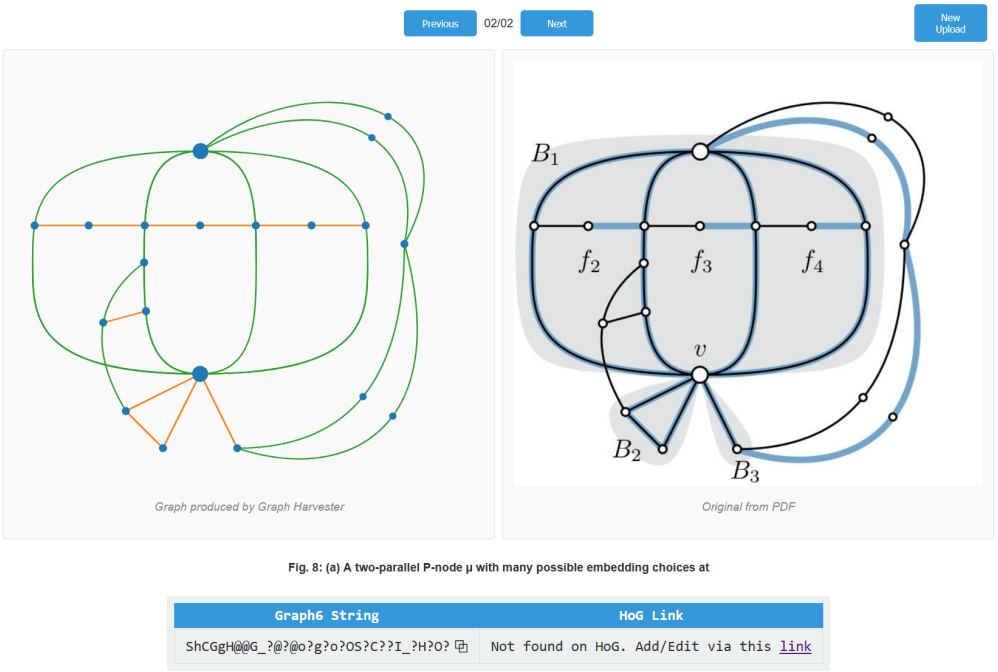



**Fig. 6:** Output of Graph Harvester for Figure 8(b) of the paper [10].

## 4 Results

The steps described in the previous section are combined in a website that is able to detect graphs and their drawings from vector graphics in PDF files. It is not only able to detect straight-line drawings, but also drawings with curved edges. Furthermore, the tool is user-friendly and does not require specific software or hardware from the end user. Its integration with the House of Graphs allows the user to easily add graphs to the database and to identify already existing graphs. Graph Harvester was able to extract 293 potentially interesting graphs (with at least seven vertices and maximum vertex degree of at least 3) from figures in papers presented at GD 2023, of which only 57 were already contained in the HoG database.

The Graph Harvester website was presented at the 32nd International Symposium on Graph Drawing and Network Visualization (GD 2024) as a Software Presentation, receiving positive feedback from the graph drawing community. Furthermore, we plan to use and extend the website to create a dataset of graphs and their drawings published in publications from previous volumes of the Graph Drawing conference.

## 5 Conclusion

This work resulted in the development of a user-friendly website capable of detecting graphs from PDFs. Our website is not only capable of extracting straight-line drawings but also able to detect edges comprising curves. It was received well by the graph drawing community when presented at GD'24.

However, the underlying detection mechanisms should be made more robust, e.g., to also handle drawings where the components of edges do not end in the same point but overlap.

As further future work, this website may be integrated into the House of Graphs website. Additionally, a dataset of graphs and their drawings may be created by the authors using the underlying pipeline.

# Appendices

## A Example for Illustrating the Importance of Regained Vertex Segments

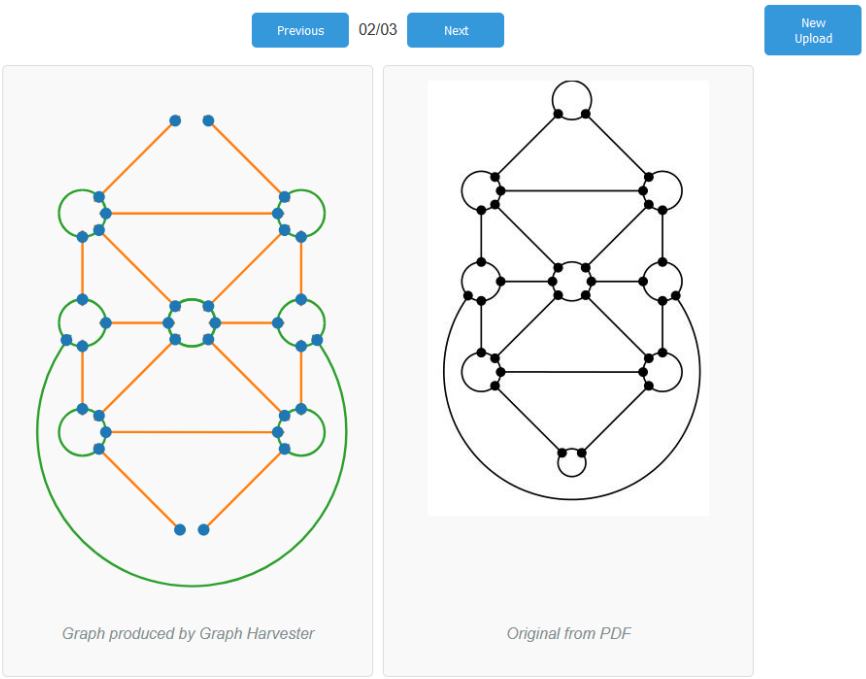



**Fig. 6. Vertices of the graph G (a) are replaced by cycles (b). A non-intersecting cycle**

| Graph6 String | HoG Link |
| --- | --- |
| ahSGGCD?G?_@?D??_?H?@??S??G?GG??C??@?_?G_?G_??C???@????_G??g???A????_????G?O??G??A?S????C??G??? | Not found on HoG. Add/Edit via this link |

**Fig. 7:** Output of Graph Harvester for Figure 8(b) of the paper [11]. Here, filtering circles based on their size is essential to extract all vertices correctly. Furthermore, the importance of reconsidering the segments of discarded vertices is illustrated in this example. Without it, the big green circles would be rightfully treated as vertex candidates and then discarded because they contain other vertex candidates. When reconsidering the circles' segments, they are correctly interpreted as edges.

## References


[1] G. Brinkmann, K. Coolsaet, J. Goedgebeur, and H. Mélot, "House of Graphs: A database of interesting graphs," *Discrete Appl. Math.*, vol. 161, no. 1–2, pp. 311–314, 2013. DOI: https://doi.org/10.1016/j.dam.2012.07.018.

[2] C. Auer, C. Bachmaier, F. J. Brandenburg, A. Gleißner, and J. Reislhuber, "Optical graph recognition," in *International Symposium on Graph Drawing*, ser. LNCS, Springer, vol. 7704, 2013, pp. 529–540. DOI: https://doi.org/10.1007/978-3-642-36763-2_47.

[3] K. Coolsaet, S. D'hondt, and J. Goedgebeur, "House of Graphs 2.0: A database of interesting graphs and more," *Discrete Appl. Math.*, vol. 325, pp. 97–107, 2023. DOI: https://doi.org/10.1016/j.dam.2022.10.013.

[4] *Description of graph6, sparse6 and digraph6 encodings*, http://users.cecs.anu.edu.au/~bdm/data/formats.txt, Accessed: 2024-05-21.

[5] B. D. McKay, "Practical graph isomorphism," *Congressus Numerantium*, vol. 30, pp. 45–87, 1981.

[6] B. D. McKay and A. Piperno, "Practical graph isomorphism, II," *Journal of Symbolic Computation*, vol. 60, pp. 94–112, 2014. DOI: https://doi.org/10.1016/j.jsc.2013.09.003.

[7] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using networkx," *Proceedings of the 7th Python in Science Conference (SciPy2008)*, vol. 2008, pp. 11–15, 2008.

[8] SageMath Development Team, *Sagemath, the Sage mathematics software system (version 9.8)*, https://www.sagemath.org, 2023.

[9] S. Kempf, M. Krug, and F. Puppe, "KIETA: Key-insight extraction from scientific tables," *Appl. Intell.*, vol. 53, pp. 9512–9530, 2023. DOI: https://doi.org/10.1007/s10489-022-03957-8.

[10] S. D. Fink, M. Pfretzschner, and I. Rutter, "Parameterized complexity of simultaneous planarity," in *International Symposium on Graph Drawing and Network Visualization*, ser. LNCS, Springer, vol. 14466, 2023, pp. 82–96. DOI: https://doi.org/10.1007/978-3-031-49275-4_6.

[11] P. de Nooijer, S. Terziadis, A. Weinberger, *et al.*, "Removing popular faces in curve arrangements," in *International Symposium on Graph Drawing and Network Visualization*, ser. LNCS, Springer, vol. 14466, 2023, pp. 18–33. DOI: https://doi.org/10.1007/978-3-031-49275-4_2.