Calibration in optical graph recognition

Christopher Auer, Christian Bachmaier, Franz J. Brandenburg,
Andreas Gleißner, Josef Reislhuber

University of Passau, 94030 Passau, Germany.
{auerc, bachmaier, brandenb, gleissner, reislhuber}@fim.uni-passau.de

Abstract Graph drawing is the process of transforming the topological structure of a graph into a graphical representation. Primarily, it maps vertices to points and displays them by icons, and it maps edges to Jordan curves connecting the endpoints. Optical graph recognition (OGR) is the inverse and transforms the digital image of a drawn graph into its topological structure. It consists of four phases: preprocessing, segmentation, topology recognition, and postprocessing. OGR is based on established digital image processing techniques. Its novelty is the topology recognition where the edges are recognized with emphasis on the attachment to their vertices and on edge crossings. Our prototypical implementation OGR$^{\text{up}}$ shows the effectiveness of the approach and produces a GraphML file, which can be used for further algorithmic studies and graph drawing tools. It has been tested both on hand made graph drawings and on drawings generated by graph drawing algorithms. Here we report on experiments for the calibration of parameters, which are critical for topology recognition.

Keywords: optical graph recognition, graph topology, inverse of graph drawing, calibration

1 Preliminaries

We consider undirected graphs $G = (V, E)$ with sets of vertices V and edges E, their drawing and their recognition. A *drawing* of G maps vertices to graphical objects like discs, rectangles, or other shapes in the plane. The edges are mapped to Jordan curves connecting its endpoints. For convenience, we speak of vertices and edges when they are elements of a graph, of its drawing, and in a digital image of the drawing. A *port* is the point of an edge which meets the vertex. Each edge has exactly two ports. Note that this is only true if an edge does not cross another vertex. Otherwise, it is hard to recognize the graph correctly both by a human and by OGR. An *(edge-edge-) crossing* is a region of points where two or more edges cross. Crossings pose the main problems in optical graph recognition.

2 Optical Graph Recognition

OGR is divided into the four phases: preprocessing, segmentation, topology recognition, and postprocessing. The input of the first phase is a drawing of a graph G as a digital image. The output is a description of G in GraphML file format [2]. From an information theoretic point of view, a digital image with a size of several MB is reduced to a GraphML file of only a few kB.

We briefly describe how each phase is realized in our prototypical implementation OGR$^{\text{up}}$, see Fig. 1. All phases but the topology recognition use standard image processing techniques. The purpose of the preprocessing phase is to separate the pixels of the background from the pixels of the drawn graph. Therefore, the image is binarized and then improved by a morphological noise reduction method [3]. In a nutshell, segmentation identifies the vertices in the binary image. More precisely, for each object pixel it determines whether it belongs to a vertex or to an edge. The

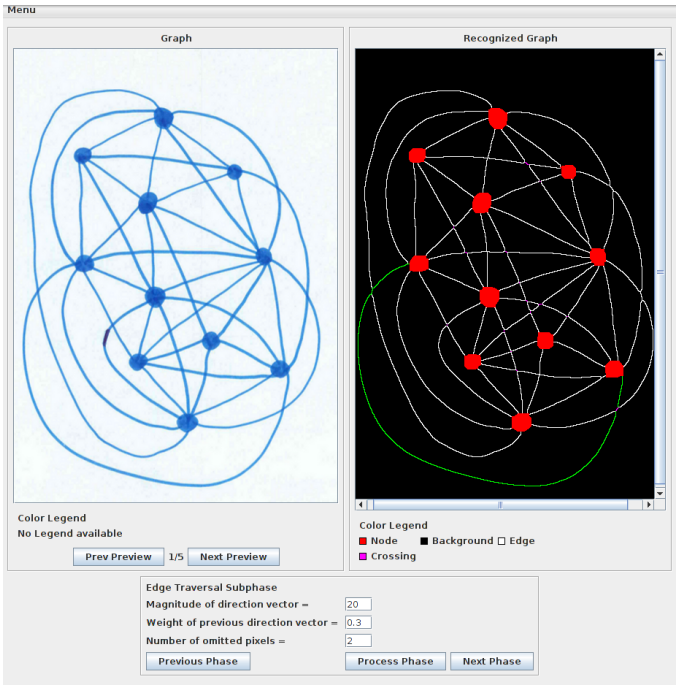

Figure 1: Graphical user interface of OGR$^{\text{up}}$ with a drawing of a graph and the recognized graph.

output is a *ternary image* with three colors for background, vertex and edge pixels. Note that, depending on the shape of the vertices, different methods have to be applied.

Topology recognition is the particularity of our approach. It is divided into *skeletonization*, *edge classification* and *edge traversal*. Skeletonization performs morphological thinning to obtain a 4-connected skeleton. The edge classification subphase classifies the pixels of the skeleton based on their 4-neighborhood and the pixels recognized as vertex pixels in the segmentation phase. The outcome are *edge pixels*, *crossing pixels*, *vertex pixels*, *port pixels* and *miscellaneous pixels*.

The classification of pixels allows us to identify *edge sections* in the image. An edge section entirely consists of edge pixels up to both endpoints. Every edge section is a 4-region. Based on the two endpoints, we classify the sections in three categories as follows. In *trivial sections* both endpoints are port pixels, e. g., edge section "1" in Fig. 2. In *port sections* one endpoint is a port pixel and the other is a crossing pixel, e. g., edge sections "2", "3", "5", and "6" in Fig. 2. Finally, in *crossing sections* both endpoints are crossing pixels, e. g., section "4" in Fig. 2.

While experimenting with our edge classification approach [1] we observed that if two edges cross, then the result was often a short intermediate crossing section like "4" in Fig. 2. This is problematic for the subsequent edge traversal subphase. Our solution is to first detect crossing sections like "4" in Fig. 2 and then to interpret them as part of a crossing. We choose a parameter γ for the minimum size of a crossing section and interpret each smaller crossing section as part of a single crossing. For a "good" graph recognition γ has to be chosen appropriately.

Trivial sections directly connect two vertices without interfering with any other edge. For every trivial section we directly obtain an edge, e. g., section "1" in Fig. 2. In contrast, port and crossing sections need a more elaborate treatment as these sections are caused by crossings. In the edge traversal phase, we merge port and crossing sections "adequately" to edges. In Fig. 2, sections "2", "4" and "6" are merged to one edge as well as "3", "4" and "5". We start the traversal always at a port section and traverse it until we find a pixel that is common to two or more edge sections. At this point we determine the adjacent edge section which most probably belongs to the current section using *direction vectors*. The direction vector of an edge section $\overrightarrow{e}$ is a two dimensional vector, which defines the direction of e. An example is illustrated in Fig. 2, where we start at port

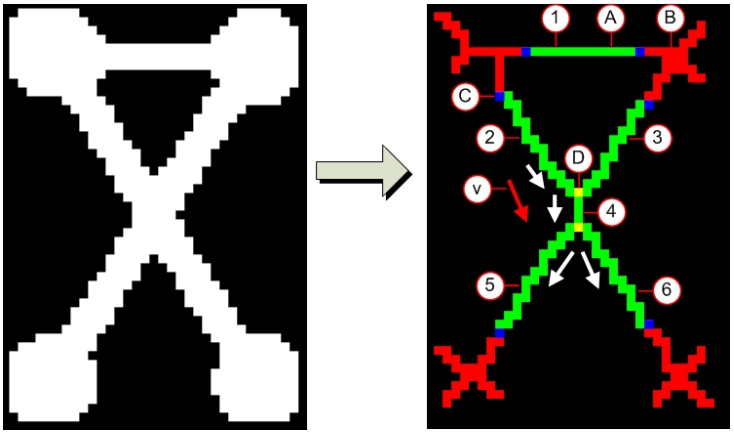

Figure 2: A binary image of a graph and the result of edge classification. An edge pixel is marked with “A”, a vertex pixel with “B”, a port pixel with “C”, and a crossing pixel with “D”. The edge sections are marked with numbers and v is a direction vector.

section “2”, reach crossing “D” and then compare the directions of port section “3” and crossing section “4” with the direction of “2”. In this case “4” is chosen, since its direction is most similar to the direction of “2”. It is important that the direction vector of an edge section does not take the whole section into account. That is, direction vector $\overrightarrow{2}$ is not computed from “C” to “D”, but only from the immediate area around the crossing. This is necessary for graphs with non-straight edges, where the edges are arbitrary Jordan curves. Hence, when approaching an edge crossing, only the local direction must be used to determine the most likely subsequent edge section.

Continuing with the example from Fig. 2, we may determine an edge consisting of “2” followed by “4”. Next both sections “5” and “6” are suitable follow-up sections if the direction vector of “4” is considered. To resolve this, we additionally take the direction vector of the preceding edge section (if existent) into account as indicated by direction vector “v” in Fig. 2. Let e_i be the edge section for which the subsequent section must be determined, $\overrightarrow{pq}$ the direction vector of e_i, and e the current edge of which e_i is part of. If e consists of more than one edge section, we take the predecessor e_{i-1} of e_i in e, determine the direction vector of e_{i-1} denoted by $\overrightarrow{rs}$, and compute the direction vector of e_i as $\overrightarrow{pq}' = \overrightarrow{pq} + \alpha \cdot \overrightarrow{rs}$ with $0 \leq \alpha \leq 1$. The parameter α is called *preceding section weight*. This parameter must be chosen appropriately. The reason for the α weighting is to reduce the influence of $\overrightarrow{rs}$ on the final direction vector $\overrightarrow{pq}'$. The relaxation is to reduce the influence of prior edge sections to avoid unsolicited results. Due to this modification, section “6” is chosen as the succeeding section of “4”, and since “6” is a port section, we have recognized an edge of the graph consisting of “2”, “4” and “6”. In the same way, the edge consisting of “3”, “4”, and “5” is recognized. With the edge consisting of “1” we now have recognized the topology of the input with four vertices and three edges.

The postprocessing phase concludes OGR and includes procedures that use the topological structure as input. Possible tasks are the assignment of coordinates to the vertices obtained in the segmentation phase, adding support points with coordinates to the edges, the attachment of labels recognized in the preprocessing phase, the recognition of edge directions, the assignment of colors, and the transformation to file formats. In OGR$^{\text{up}}$ we assign coordinates to the recognized vertices, add support points to each edge such that it resembles its counterpart from the original image and transform the data into the GraphML file format.

For maximum flexibility we designed and implemented a general module system as a backbone of the OGR$^{\text{up}}$ architecture. Basically, a module is an (isolated) subroutine with defined input and output slots transforming the input to the output. Multiple modules can be connected such that the input slot of a module is either bound to a constant value or fed by the output slot of another module. The modules are assembled on a high level of abstraction. Thus, the architecture

admits multiple realizations of an OGR phase by different algorithms, a module decomposition into replaceable subcomponents, and a supply of modules by data produced in earlier phases without interfering with steps in between. The system automatically generates boilerplate-code for the transfer of data between different modules, determines an execution order, and can parallelize the execution of independent modules. The implementer of a specific module can focus on the algorithmic aspects. Currently OGR$^{\text{up}}$ consists of 8 modules.

3 Experiments with OGR$^{\text{up}}$

A screenshot of our prototype OGR$^{\text{up}}$ is shown in Figure 1. OGR$^{\text{up}}$ assumes that the vertices are drawn as filled shapes, e. g., circles or rectangles, and have approximately the same size. The edges are curves of a width significantly smaller than the diameter of the vertices. They should be contiguous and should exactly end at the vertices.

If these conditions are met, then OGR$^{\text{up}}$ recognizes graphs very well with a rate of almost 100%, in particular if the drawings are plane or have "good" crossings [1]. Plane drawings have no crossings at all and crossings of edges are "good" if edges cross at large angles and several edges do not cross in a small region. This parallels recent empirical evaluations [4]. If the crossings are not "good", OGR$^{\text{up}}$ may erroneously swap the edges at a crossing as if they would touch or it may classify such a region as a new vertex. Therefore, OGR$^{\text{up}}$ puts particular emphasis on these situations with the parameter α for the weighting of the preceding section and the parameter γ for the minimum size of a crossing section. As often in digital image processing approaches, the results of OGR$^{\text{up}}$ depend on the careful adjustment of the relevant parameters. Here we have stressed this point for a confidence rate by a majority decision.

Our test suite consists of 92272 drawn graphs, which are obtained from 11534 Rome[1] graphs (undirected graphs with 10 to 100 nodes), where each graph was drawn 8 times with a spring embedder algorithm [5]. A spring embedder is a randomized algorithm where each run likely produces a different drawing. Each drawn graph was checked 12 times by OGR$^{\text{up}}$ with different values for α and γ. A majority decision was taken as the result of the graph recognition by OGR$^{\text{up}}$. This result was compared with the correct input graph up to graph isomorphism. Here we achieved a correct recognition rate of 89.68%.

It seems that we have already found good default values for α and γ for this test suite, since the majority decision rule has led to a recognition rate which is inferior to our standard test with one run. There the recognition rate was 93.24% [1]. In all examples, the running time of OGR$^{\text{up}}$ for the recognition of a single graph was less than 10 seconds, however, there is room for improvements. The maximum resolution of the drawings was approximately 2600×2600 pixels (or 360kB PNG images).

References

[1] Auer, C. and Bachmaier, C. and Brandenburg, F. J. and Gleißner, A. and Reislhuber, J.: Optical Graph Recognition. In Proc. Graph Drawing 2012, LNCS (2013) to appear

[2] Brandes, U. and Eiglsperger, M. and Herman, I. and Himsolt, M. and Marshall, M. S.: GraphML Progress Report: Structural Layer Proposal. In Proc. Graph Drawing 2001, LNCS 2265 (2002), 501–512

[3] Gonzalez, R. C. and Woods, R. E.: Digital Image Processing. Prentice Hall (2002)

[4] Huang, W. and Eades, P. and Hong, S.: Beyond Time and Error: A Cognitive Approach to the Evaluation of Graph Drawings. In Proc. BELIV 2008, 3:1–3:8

[5] Kaufmann, M. and Wagner, D.: Drawing Graphs. LNCS 2025 (2001)

[1] http://www.graphdrawing.org/data.html