Finding faces in a planar embedding of a graph

[Extended Abstract]

Sophie Schneider * and Ivo F. Sbalzarini
MOSAIC Group
Chair of Scientific Computing for Systems Biology, Faculty of Computer Science, TU Dresden
& Max Planck Institute of Molecular Cell Biology and Genetics, Dresden, Germany
& Center for Systems Biology Dresden, Pfotenhauerstr. 108, Dresden, Germany.
{ososchnei,ivos}@mpi-cbg.de

ABSTRACT
Planar graphs are frequently used to analyze morphological properties of networks occurring in complex systems. In these networks, an important task is to successfully extract the faces. We introduce a versatile algorithm to extract the faces of a planar embedding of a graph. With a focus on developmental biology and urban street patterning, we apply the algorithm to extracting cells from tissue graphs and areas of building density from urban street networks. We present the theory as well as other possible situations where the algorithm can be very useful.

Categories and Subject Descriptors
F.5.1 [Theory of computation]: Design and analysis of algorithms—Graph algorithms analysis; L.4.1 [Applied Computing]: Life and medical sciences—Computational biology

General Terms
ALGORITHMS

1. INTRODUCTION
The concept of graphs is widely used in economic, social, and biological systems. It provides us with a tool to illustrate and analyze networks such as transportation networks, relationships between individuals, and biochemical pathways. Planar graphs have been extensively used to model systems ranging from urban street patterns [11] to epithelial tissue formations [1] and vein patterning of plant leaves [5, 13]. In this work, we present an algorithm that can be used to identify single cells in an epithelial tissue graph, areas of building density in an urban street map, and varying vein patterns in plant leaves. All of these examples can be formalized as finding faces in a planar embedding of a graph. To our knowledge, there are only few easily accessible algorithms to find faces in planar graphs, e.g., the planar face traversal of the Boost Graph Library [10]. Our algorithm is implemented in MATLAB [6], which makes it easier to use and integrate in MATLAB-based workflows.

First, we present the theoretical background of the algorithm and then explain the algorithm in detail. We then illustrate the performance of the algorithm on three artificial graphs, and demonstrate its application to real-world networks.

2. THEORETICAL BACKGROUND
A planar graph \( G = (V, E) \) with vertices \( V \) and edges \( E \) is a graph that can be drawn in a two-dimensional (2D) plane such that no edges cross [12]. We refer to such a drawing as a planar embedding of the graph. In our algorithm, we assume that the planar embedding is known, thus we know the 2D coordinates of the vertices. Areas that are enclosed by edges are called faces. This also includes the outer face, which is bounded by all edges at the border of the embedding and is infinitely large.

The number of faces \( |F| \) (including the outer face) in a planar graph can be calculated using Euler’s formula [12]:

\[
|F| = 2 - |V| + |E|, \tag{1}
\]

where \( |V| \) is the number of vertices in the graph and \( |E| \) the number of edges. We use this formula to verify that we found all faces in a large graph, such as for example in Fig. 7.

We design our algorithm for undirected graphs, but it can be extended to directed graphs. Whether the graph is weighted or unweighted is of no significance to our problem. Furthermore, we only consider planar embeddings where all edges are straight lines. According to Fáry’s theorem [2], every planar graph can be embedded in this way, such that this assumption is not limiting.

3. ALGORITHM
The general idea of our algorithm is to iterate over all vertices of the graph and enumerate all faces that contain this vertex. From each vertex, the edges are traversed in a certain orientation (i.e., clockwise or counter-clockwise) in order to find all faces that contain the initial vertex. To determine the edges we need to traverse, we compute the orientation and the angle between the current edge and the next candidate edge, as described next.

* Corresponding author
The orientation is easily computed using the cross product between the incoming edge (vector \( \vec{a} \), Fig. 1) and the vector \( \vec{c} \) (Fig. 1) between the starting vertex (A) and the ending vertex (C):

\[
\vec{a} \times \vec{c} = \begin{pmatrix} a_x & c_x \\ a_y & c_y \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & a_x c_y - a_y c_x \end{pmatrix}.
\]

In case of a clockwise orientation of \( \vec{a} \) and \( \vec{b} \), the cross product is positive, whereas it is negative for a counterclockwise orientation. If more than one candidate edge are in counterclockwise orientation, we take the one with the smallest angle. If there are only clockwise edges, we take the one with the largest angle.

**Algorithm 1** Find faces

1: for all \( v \in V \) do
2: for all adjacent \( v_{adj} \) to \( v \) do initialize visit
3: while \( v \not\in \text{visit} \) do
4: if |\( \text{visit} \)| == 3 and counter-clockwise then
5: for all candidate vertices \( v_{cand} \) do
6: if |\( v_{cand} \)| == 1 then add \( v_{cand} \) to visit
7: else calculate orientation \( \alpha \) and angle \( \alpha \)
8: find \( v_{cand} \) leading to counterclockwise orientation \( v_{ccw} \)
9: if |\( v_{ccw} \)| == 1 then add \( v_{ccw} \) to visit
10: else if |\( v_{ccw} \)| > 1 then add \( v_{cand} \) with minimal \( \alpha \) to visit
11: else if |\( v_{ccw} \)| == 0 then
12: if any \( v_{cand} \) is collinear then
13: add \( c_{cand} \) to visit
14: else add \( v_{cand} \) with maximal \( \alpha \) to visit
15: return

The overall algorithm is summarized in Alg. 1 and illustrated in a small working example in Fig. 2.

**Runtime and space complexities**

The runtime of our algorithm depends on \( |V| \) and on the topology of the graph. In the worst-case where the graph consists of a single face, the time complexity would be \( O(|V|^2) \). However, in practice this is a trivial case. In an average case, the expected runtime of our algorithm is \( O(|V| \cdot k) \) where \( k \) is a connectivity-related constant. The space complexity is dominated by the number of elements in the adjacency matrix, \( O(|V|^2) \). The timings in Tab. 1 show a linear dependency on \( |V| \).

**Validation**

We validate the algorithm by comparing the number of faces found to the theoretical number of faces from Eq. 1. We visualize the results by plotting all faces in the same semitransparent color. If one face appears darker than the other faces, or without any color, the algorithm has detected it twice to not at all, respectively. For this validation we use the three small synthetic graphs shown in Fig. 4, and the five real-world graphs shown in the Applications section. The algorithm worked well on all tested graphs within the limitations stated next.

**Limitations**

The present algorithm correctly finds all faces in a planar embedding of a graph as long as all faces are simple polygons. A simple polygon encloses an area of finite size, has exactly two edges meet at each vertex, and has a total number of vertices that is equal to the total number of edges \([3]\). According to this limitation, the present algorithm cannot identify faces in which any vertex appears more than once. Also, according to these limitations, vertices with a degree of less than two have to be pruned from the graph before applying the present algorithm. This is illustrated in the example in Fig. 3.

**4. APPLICATIONS**

We demonstrate the application of the present algorithm to three synthetic examples (shown in Fig. 4) and three examples of real-world planar graphs: urban street maps, plant leaf vein patterns, and epithelial tissue cell patterns. In all cases we measure the runtime in seconds on a MacBook Pro with 2.3 GHz Intel Core i7 processor and 16 GB DDR3 RAM. The algorithm is implemented and run in MATLAB R2013b. The input graphs and results of the three synthetic examples are shown in Fig. 4. For these small examples, the runtimes were in the range of a few tens of milliseconds (Tab. 1).

For three real-world test cases we first reconstructed the input graphs from images by binarizing the image and subsequently skeletonizing the binary image. Image binarization and skeletonization are conducted in Fiji [9] using an Otsu threshold [7] to binarize the image and a thinning algorithm [14] to skeletonize the binary mask. The input graphs are then constructed from the skeletonized images as described in Appendix A. All vertices of degree less than two are removed from the graphs before running the present algorithm.
4.1 Urban streets maps
We use cropped parts of two urban street maps (New York City and London), taken from Ref. [4]. Finding faces in these networks corresponds to finding areas of building density in cities, which can be used to optimize navigation or planning. The results are shown in Fig. 5.

4.2 Leaf veins
We apply our algorithm to the vein patterns of different plant leaves. The images were taken from Ref. [8]. The results are shown in Fig. 6 and can be used to analyze the morphological patterning of leaves across species, and to study tissue profusion.

4.3 Epithelial tissue
Epithelial tissues are one of the four basic types of animal tissues, lining the cavities and surfaces of structures throughout our body (e.g., lungs, intestines, eyes). A well-studied model system for epithelia is the imaginal wing disc of D. melanogaster (fruit fly) embryos. This tissue, which later develops into the fly’s wing, consists of a single layer of densely packed cells. Fig 7 shows a microscopy image of a fly wing disc in the left panel, where all cell membranes are brightly stained. Applying the present algorithm leads to the result shown in the right panel, identifying all cells in the tissue. This enables, e.g., computing statistics of cell size, growth rates, or neighborhood relations between cells.

5. CONCLUSIONS
We presented a novel algorithm for finding faces in a graph with known two-dimensional embedding. The algorithm is applicable to any such graph, as long as all faces are simple polygons and no vertices of degree less than two exist. We have validated and timed the algorithm on synthetic graphs, and demonstrated its practical application in three real-world cases to extract cells in an epithelial tissue, areas of building density in urban street maps, and vein patterns in plant leaves.

Similar to the Boost graph library, our algorithm on average has linear time complexity. Our code is designed for non-computer scientists and can be directly included in native MATLAB implementations. Future work includes a parallelization of our algorithm to handle very large graphs and reducing the memory requirements by using more efficient data structures, e.g., adjacency lists. The MATLAB implementation of the presented algorithm will be freely available from the web page of the authors upon publication of the paper.

Figure 7: Left panel: Raw microscopy image of a fly wing disc (image courtesy of the Dahmann lab, TU Dresden). Right panel: tissue graph with colored faces corresponding to individual biological cells.

6. ACKNOWLEDGMENTS

We thank the members of the MOSAIC Group and Ömer Demirel (S3IT, University of Zurich) for helpful discussions, as well as Sui Liyuan (Dahmann group, TU Dresden) for providing the raw image of the epithelial tissue case. This work was supported in parts by the German Federal Ministry of Research and Education (BMBF) under funding code 031A099.

7. REFERENCES


APPENDIX

A. 2D GRAPH CONSTRUCTION

We construct planar embeddings of graphs from a skeletonized image by first treating each non-zero pixel in the image as a vertex. Then, we detect pixels with more than one non-zero neighbor and label them as branching points. Using depth-first search [12] we traverse the graph starting from each branching point until a next branching point is reached. If there exists a path between two branching points, a corresponding edge is inserted and all vertices along the path are deleted. The branching points correspond to the vertices in the final graph.

| graph               | |V| | |E| | |F| | time [sec] |
|---------------------|-----|----|-----|-----|-----|-----|-----|
| butterfly           | 5   | 6  | 2   |     |     | 0.02|
| regular grid        | 9   | 12 | 4   |     |     | 0.03|
| pacman              | 6   | 7  | 2   |     |     | 0.02|
| NYC street map      | 114 | 192| 79  |     |     | 1.35|
| London street map   | 227 | 332| 106 |     |     | 1.35|
| B.victoriae leaf    | 1208| 1754| 547 |     |     | 8.87|
| W. floribunda leaf  | 1524| 2087| 564 |     |     | 11.25|
| D. melanogaster wing| 2951| 4442|1493 |     |     |24.29|

Table 1: Timings for test cases. The increase in runtime is linear in |V|. From top to bottom, the space requirements of the graphs increase. From ‘New York City map’ downward, the data cannot fit in the cache, which causes a jump in the timings.