Title
Graph Drawing Metrics and Representations with Applications

Author
Johnson, Timothy Underwood

Publication Date
2018

UNIVERSITY OF CALIFORNIA,
IRVINE

Graph Drawing Metrics and Representations with Applications

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Science

by

Timothy Underwood Johnson

Dissertation Committee:
Professor Michael T. Goodrich, Chair
Professor David Eppstein
Professor Michael Dillencourt

2018

DEDICATION

To my wife Xiao; without her encouragement and support this dissertation would have remained unfinished.

TABLE OF CONTENTS

LIST OF FIGURES

Page

ACKNOWLEDGMENTS

First, I would like to thank my advisor Michael Goodrich for his mentorship. I would also like to thank David Eppstein and Michael Dillencourt for serving on my defense committee, and Sandy Irani and James Jones for serving on my advancement committee.

I would like to thank my coauthors: Muhammad Jawaherul Alam, Juan Jose Besa, Giordano Da Lozzo, William Devanny, David Eppstein, Michael T. Goodrich, and Manuel Torres. I would also like to thank Matthew Might, Michael Adams, William Byrd, and Guannan Wei for their assistance and guidance in the DARPA STAC project.

I would like to thank my fellow graduate students in the Center for Algorithms and Theory of Computation. Our community has been a source of entertainment, inspiration, collaboration, and valuable feedback. In particular, I would like to thank William Devanny and Nil Mamano for coordinating our theory tea time.

I would like to thank the Donald Bren School of Information and Computer Science for their funding support. I would like to thank DARPA for supporting this work under agreement no. AFRL FA8750-15-2-0092. This work was also supported in part by the U.S. National Science Foundation under grants 1228639, 1526631, 1618301, and 1616248, 1815073. The views expressed in this dissertation are those of myself and my coauthors and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

CURRICULUM VITAE

Timothy Underwood Johnson

EDUCATION

Doctor of Philosophy in Computer Science **2018**
University of California, Irvine *Irvine, California*

Bachelor's of Science in Mathematics **2013**
California Institute of Technology *Pasadena, California*

Associate's of Science in Mathematics **2010**
El Camino College *Torrance, California*

REFEREED CONFERENCE PUBLICATIONS

Low Ply Drawings of Trees and 2-Trees **August 2018**
Canadian Conference on Computational Geometry

Optimally Sorting Evolving Data **July 2018**
International Colloquium on Automata, Languages and Programming

Quadratic Time Algorithms Appear to be Optimal for Sorting Evolving Data **January 2018**
Algorithm Engineering and Experiments

Square-Contact Representations of Partial 2-Trees and Triconnected Simply-Nested Graphs **December 2017**
International Symposium on Algorithms and Computation

J-Viz: Sibling-First Recursive Graph Drawing for Visualizing Java Bytecode **October 2016**
IEEE Symposium on Visualization for Cyber Security

ABSTRACT OF THE DISSERTATION

Graph Drawing Metrics and Representations with Applications

By

Timothy Underwood Johnson

Doctor of Philosophy in Computer Science

University of California, Irvine, 2018

Professor Michael T. Goodrich, Chair

Much of graph drawing is based on drawing graphs as node-link diagrams, in which vertices are represented by points and edges are drawn as lines between the points. We first investigate a newly developed metric for measuring the quality of such a diagram called ply number, which is inspired by studying road networks. We then consider contact representations, in which vertices are represented by geometric shapes and edges exist as intersections along the boundaries of these shapes. Lastly, we conduct experiments in the use of graph drawing in visualizing the structure of computer programs.

The ply number of a node-link drawing is defined as follows. For each vertex v, which is associated with a point in the plane, a disk is drawn centered on v with a radius of alpha times the length of the longest edge incident to v. The ply number is the maximum number of disks that overlap at a single point. We identify values of alpha for which a bounded-degree tree can be drawn with no two ply disks overlapping. We also show that for the customary value of alpha=1/2, all trees can be drawn with logarithmic ply number, with an area that is polynomial for bounded-degree trees. Lastly, we give a lower bound for the ply number of drawings of a specific class of 2-trees.

We then study proper square-contact representations of planar graphs, in which any two squares are either disjoint or share infinitely many points along their boundary. We charac-

terize the partial 2-trees and the triconnected cycle-trees allowing for such representations. We also study square-contact representations of general triconnected simply-nested graphs with respect to their outerplanarity index.

Finally, we describe a graph visualization tool for visualizing Java bytecode. Our tool, which we call J-Viz, visualizes connected directed graphs according to a canonical node ordering, which we call the sibling-first recursive (SFR) numbering. We show through several case studies that the canonical drawing paradigm used in J-Viz is effective for identifying potential security vulnerabilities and repeated use of the same code in Java applications.

Chapter 1

Introduction

In this dissertation, we consider the problem of drawing graphs algorithmically. Graphs are a mathematical abstraction for representing information by a set of objects and a binary relation on those objects. Since first being used by Euler to study the bridges of Königsberg (Figure 1.1), graphs have become an extremely useful way to represent information. For example, a graph could represent a social network, in which the objects are people, and the binary relation is friendship between pairs of people. Alternatively, a graph could be used to represent a road network, in which the objects are intersections, and the elements of the relation correspond to roads that connect the intersections.

Formally, an undirected graph G consists of a set V of vertices, and a set E of edges, each edge being an unordered pair of vertices in V. (In constrast, for directed graphs, the edges are ordered pairs.) In graph drawing, we try to display a graph visually so that its structure can be clearly seen. The most common style of graph drawing is the node-link diagram, in which each vertex is represented by a point, and each edge by a line between the points corresponding to the vertices that it connects.

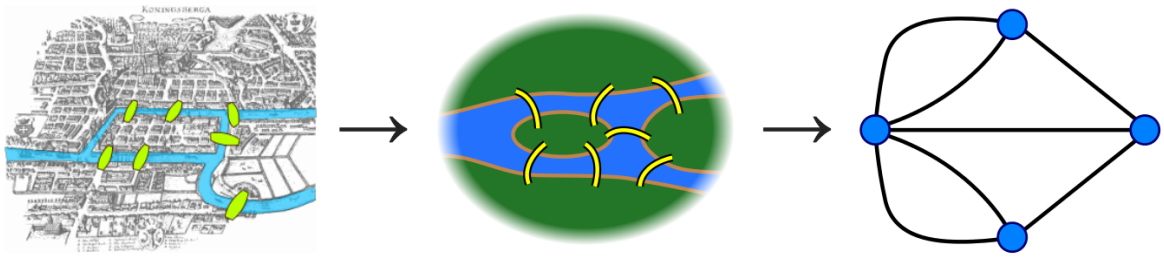

Figure 1.1: The seven bridges of Königsberg, and a node-link drawing of their abstraction as a graph [64].

1.1 Graph drawing metrics

After defining the style in which we will present a graph drawing, it is natural to consider metrics for judging the quality of the drawing. Since the end goal is to produce drawings that people will understand and appreciate, graph drawing necessarily requires a choice of aesthetic criteria. We would then like to find algorithms to construct drawings that optimize these criteria.

Many criteria have been identified as aesthetically pleasing and useful for understanding graphs. For node-link diagrams, these include minimizing the number of crossings between edges and minimizing the number of bends in the edges [52]. But these metrics were only tested for graphs with a few dozen vertices. For large graphs, a more common approach is to try to minimize the stress of a drawing [29, 35, 44], which measures whether the distance between each pair of nodes is close to the desired distance.

In Chapter 2, we explore a new metric for assessing straight-line node-link diagrams. This work follows the paradigm of drawing graphs to mimic the properties of road networks [28, 50, 51]. This new metric has been found to be well-satisfied by road networks [22]. It requires the vertices in our drawing to be well-separated at multiple scales, and encourages the edges adjacent to each vertex to have similar lengths. It also is closely related to stress, as several

algorithms that are designed to minimize stress also tend to produce drawings with low ply number [15].

For our results, we extend the study of which classes of graphs can be drawn with low ply number. In particular, we give two different drawing algorithms that provide upper bounds on the ply number of trees, and give a new lower bound on the ply number of 2-trees.

1.2 Contact representations

Instead of node-link diagrams, we can also represent vertices as simple closed geometric shapes, and represent edges by intersections between the shapes. In some variants the interiors of the nodes are allowed to intersect, but we will study the simpler case in which intersections are only allowed along the boundaries. Such a drawing is called a *contact representation* (see Figure 1.2). Ideally, a contact representation for a graph represents its information in a similar fashion to how a world map gives insight into relationships between countries.

Contact representations have been widely studied for many different shapes. For example, the famous Koebe–Andreev–Thurston theorem [59] states that a graph has a contact representation using circles if and only if it is planar (i.e., can be drawn in the plane without crossings). A far-reaching extension of this known as Schramm's monster packing theorem [55] shows that planar graphs have contact representations using translations and scalings of any smooth convex body. However, for non-smooth shapes, this algorithm may produce degeneracies, in which some of the shapes are collapsed to a single point.

Contact representations have also been studied for unit circles [8, 36], line segments [41], circular arcs [1], triangles [32], L-shaped polylines [11], and cubes [25]. However, little is known about contact representations using axis-aligned squares.

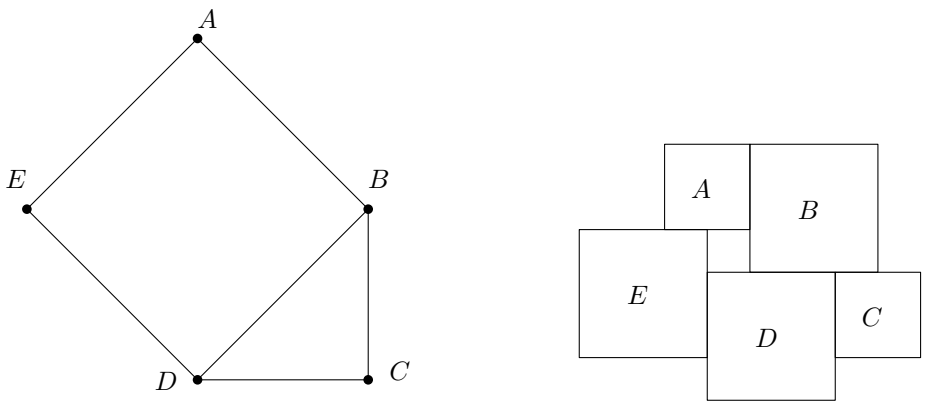

Figure 1.2: A node-link drawing of a graph, and an equivalent proper square contact representation.

In Chapter 3, we study *proper* square contact representations, in which adjacent axis-aligned squares must intersect along one of their sides, rather than just at a single point. A graph with a contact representation in this model must be planar, and must have a planar embedding in which there are no triangles that contain other vertices. We show two new families of graphs for which these conditions are sufficient, the partial 2-trees and the cycle trees. However, we also show that for general planar graphs these two conditions are not sufficient, even if we allow improper contact representations.

1.3 Software visualization

Finally, we study a specific application of graph drawing, in which we would like to interactively display the structure of a software program to allow an analyst to identify potential security vulnerabilities for algorithmic complexity attacks. The simplest way to create a graph that represents all of the program's execution paths is to create one vertex for each instruction, and add an edge from each instruction to all of the instructions that could be executed immediately after it. This graph is *sound*, since every execution path of the program has a corresponding path in the graph. But it is not *precise*, since there may be many paths in the graph that are not actually feasible for the program. Therefore, we can modify

our abstraction to use Shiver's k-CFA framework [57]. This framework is always sound, and allows us to make our static analysis more precise at the cost of potentially increasing the size of the graph.

There are also many previous systems that apply graph visualization to source code, such as for showing call graphs [17] or automatically generating documentation [61]. However, we believe that our system, which we have titled J-Viz, has several advantages. First, it analyzes code at a greater level of detail, since it parses individual instructions rather than just the call graph. Second, it allows an analyst to interact with the graph by expanding regions of interest, which surveys have found is one of the most requested features for source code visualization [6]. At the same time, our layout is chosen so that these sections that are expanded or collapsed do not affect the layout for the rest of the graph. This allows the analyst to preserve their mental map of the code, which user studies have shown to be very important [53]. Third, it uses a layout that is simple and scalable, based around finding a spanning tree of the graph. This spanning tree is selected using a hybrid between breadth-first and depth-first search that we call sibling-first recursive, which we have found to more closely match our intuitive expectations as programmers. Lastly, we use heuristics to estimate which sections of the code will have the longest runtime, and highlight these sections for the analyst.

Chapter 2

Low Ply Drawings of Trees and 2-Trees *

2.1 Introduction

A useful paradigm for drawing graphs involves visualizing them as maps or road networks, allowing a visualizer to "zoom" in and out of the graph based on known techniques that apply to maps. For example, Gansner *et al.* [28] describe a GMap system for visualizing clusters in graphs as countries with nearby clusters drawn as neighboring countries. In addition, Nachmanson *et al.* [50, 51] describe a GraphMaps system for visualizing graphs as embedded road networks, so as to leverage the drawing and zooming capabilities of a roadmap viewer to explore the graph. Thus, a natural question arises as to which graphs are amenable to being drawn as road networks.

To answer this question, we formulate a precise definition of what we mean by a graph that could be drawn as a road network. One might at first suggest that graph planarity would be a good choice for such a formalism. But the class of planar graphs includes several graph instances that are difficult to visualize as road networks, such as the so-called "nested

*Portions of this chapter are included from [33].

triangles" graph (e.g., see [20, 27, 31]). In addition, as shown by Eppstein and Goodrich [22], the class of planar graphs is not general enough to include all real-world road networks, as road networks are often not planar. For example, the California highway system alone has over 6,000 crossings. Instead of using planarity, then, Eppstein and Goodrich [22] introduce the concept of the *ply number* of an embedded graph, and they demonstrate experimentally that real-world road networks tend to have small ply. Intuitively, the ply concept tries to capture how road networks have features that are well-separated at multiple scales. The formal definition of the ply number of a graph is derived from the definition of ply for a set of disks (which captures the depth of coverage for such a set of disks) [48]; hence, the ply number of an embedded graph is defined in terms of the ply of a set of disks defined with respect to this embedding.

Let us therefore formally define the *ply number* of an embedded geometric graph. Let Γ be a straight-line drawing of a graph G. For every vertex $v \in G$, let C_v^α be the open disk centered at v and whose radius r_v^α is α times the length of the longest edge incident to v. The set of ply disks containing a point q is then $S_q^\alpha = \{C_v^\alpha \mid \|v - q\| < r_v^\alpha\}$. The *$\alpha$-ply number* of this drawing is defined as

$$\mathrm{pn}(\Gamma) = \max_{q \in \mathbb{R}^2} \|S_q^\alpha\|.$$

Usually, α is chosen in the range $(0, 0.5]$. In this range, a graph with two vertices and a single edge connecting them has ply number 1, because the ply disks for the two vertices will not overlap.

There are two natural optimization problems for the ply number when constructing a drawing of a given graph. One is to is to fix a constant ply number, and try to find a drawing that maximizes the value of α. The other is to fix a value for α, typically $1/2$, and try to find a drawing that minimizes the ply number. In this chapter, we provide new results for both of these cases.

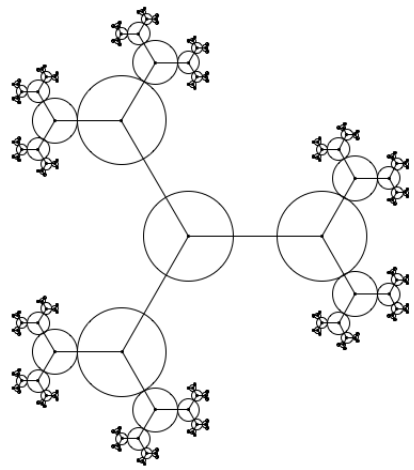

Figure 2.1: Di Giacomo *et al.*'s drawing of a binary tree with α-ply number 1, for $\alpha = \frac{1}{3}$. The edge lengths decrease by a factor of 2 at each level.

2.1.1 Previous related work

As an empirical justification of the use of ply numbers, De Luca *et al.*'s experimental study [15] found that some force-directed algorithms, including Kamada-Kawai [44], stress majorization [29], and the fast multipole method [35] all tend to produce drawings with low ply number. Their experiments also suggest that even trees with at most three children per node can have unbounded ply number when $\alpha = 1/2$.

The problem of drawing graphs with ply number equal to 1 is related to that of constructing circle-contact representations. A circle-contact representation for a graph is a collection of interior-disjoint circles, in which each circle represents a single vertex, and two vertices are adjacent if and only if their circles are tangent to one another [39,40]. Di Giacomo *et al.* [18] show that graphs with ply number 1 are equivalent to graphs with weak unit disk contact representations, which are known to be NP-hard to recognize [9]. They also show that binary trees have drawings with ply number 2 when $\alpha = 1/2$, or with ply number 1 when $\alpha \leq 1/3$. Their drawing is reproduced in Figure 2.1.

Angelini *et al.* [5] relax our definition of ply number to define the *vertex-ply* of a drawing, which is the maximum number of intersecting disks at any vertex of the drawing. Graphs with vertex-ply number 1 can then be interpreted as a new variant of proximity drawings.

In an earlier paper, Angelini *et al.* [4] show that 10-ary trees have unbounded ply number. Furthermore, they prove that 5-ary trees can be drawn with logarithmic ply number and polynomial area. The ply number of drawings of trees with between three and nine children per node remains an interesting and surprisingly daunting open problem.

2.1.2 Our results

In this chapter, we study a number of related problems concerning low-ply drawings of bounded-degree trees. We first answer an open question proposed by Di Giacomo *et al.* [18], which asks whether all trees with maximum degree Δ have 1-ply drawings for a sufficiently small α. We show in Section 2 that a simple fractal drawing pattern can achieve this when $\alpha = O(1/\Delta)$.

In Section 3, we show that all trees (not just 5-ary trees) can be drawn with logarithmic ply number, for $\alpha = 1/2$. Furthermore, the area is polynomial for trees with bounded degree. These results depend on some careful arguments about geometric configurations and fractal-like geometric constructions, as well as yet another use of the heavy-path decomposition technique of Sleator and Tarjan [58].

It is then natural to consider whether any planar graph classes larger than trees can be drawn with logarithmic ply number. In Section 4, we show that this is not the case for 2-trees, by constructing a family of 2-trees that require a ply number of $\Omega(\sqrt{n/\log n})$ for any fixed $\alpha > 0$. Previous lower bounds have only applied for planar drawings, while non-planar drawings are known to sometimes have better ply number.

2.2 1-ply Drawings

In this section, we fix our drawings to have ply number 1, and provide conditions on α such that we can construct drawings of trees of any bounded degree. At a high level, our drawings are constructed as follows. For a tree with maximum degree Δ, we divide the area around each parent vertex radially into Δ equal wedges, so that all of the angles are $2\pi/\Delta$. Then we assign each subtree to a different wedge, and draw it within that wedge. The distance from each node to its children is chosen to be a constant fraction f of its distance from its own parent. When $\Delta = 3$, this produces the drawing shown in Figure 2.1. Note that for a non-root vertex, one of the wedges will contain the edge from the parent vertex, and will not contain a subtree.

This produces a drawing that is highly symmetric, in a fashion that would produce a fractal if continued in the limit.[1] Thus, any bounded-degree tree is a subtree of this infinite tree; hence, this drawing algorithm can produce a drawing of any bounded-degree tree. Filling in the details of this construction requires setting the values of two parameters: f, the ratio between outgoing and incoming edge lengths; and α, the ratio between the radius of a ply disk for a vertex and the length of its longest incident edge. We provide constraints for the following three cases, which taken together ensure that there are no overlaps, so that the ply number of our drawings is 1. We then maximize α such that all of these constraints are satisfied.

1. Ply disks for adjacent vertices must not overlap.

2. Ply disks for vertices in separate subtrees must not overlap.

3. A ply disk for a vertex must never overlap a ply disk for one of its descendants.

[1] See Falconer [24] for further reading about fractal geometry.

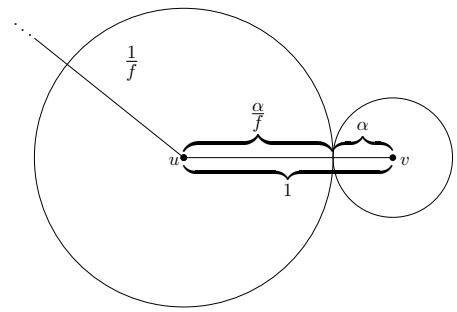

Figure 2.2: Our edges decrease by a factor of f at each level, and the ply disks have radius α times the length of the incoming edge.

It is easily verified that these three conditions are necessary and sufficient for a tree to have a 1-ply drawing.

2.2.1 Condition 1: Separate adjacent vertices

Except for the root vertex, which has no incoming edge, we proportion the lengths of the edges for each vertex as shown in Figure 2.2.

That is, taking the length of the reference edge $\overline{uv}$ as 1 (illustrated in Figure 2.2 going from parent to child in a left-to-right orientation), then, based on our definition of the α-ply number, the radius of the larger circle is α/f, the radius of the smaller circle is α, and their distance is 1. Thus, we have our first condition relating α and f.

$$\alpha/f + \alpha \leq 1 \tag{2.1}$$

$$\alpha \leq \frac{f}{1+f} \tag{2.2}$$

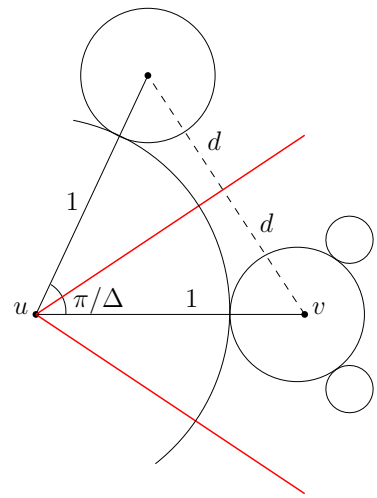

Figure 2.3: Our constraint on a wedge containing a subtree of a central vertex.

2.2.2 Condition 2: Separate subtrees with the same root

We require that the ply disks for any subtree all be contained within a wedge of angle $\theta = \frac{2\pi}{\Delta}$ around its parent vertex, where Δ is the degree. Since our wedges for each subtree are disjoint, this ensures that the ply disks for two adjacent subtrees cannot overlap.

As illustrated in Figure 2.3, the distance from a child vertex to the boundary of its containing wedge is $d = \sin\left(\frac{\pi}{\Delta}\right)$. Note also that the lengths of edges along a path in this subtree form a geometric sequence with ratio f. So the maximum distance from a child vertex to any vertex in its subtree is $\sum_{i=1}^{\infty} f^i = \frac{f}{1-f}$.

Therefore, to confine each subtree within its wedge, we must set

$$\frac{f}{1-f} \leq \sin\left(\frac{\pi}{\Delta}\right).$$

Solving for f, we get

$$f \leq \frac{\sin\left(\frac{\pi}{\Delta}\right)}{1+\sin\left(\frac{\pi}{\Delta}\right)}. \tag{2.3}$$

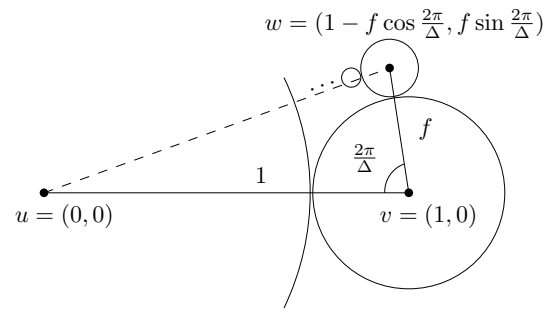

Figure 2.4: Our layout leaves a gap of angle $\theta = \frac{2\pi}{\Delta}$ for the edge from the parent vertex. The descendants on either side must not overlap their ancestors.

2.2.3 Condition 3: Separate each vertex from its descendants

Our last condition is that the ply disk for a vertex cannot overlap any of its descendants. The closest descendants will be those in the wedges on either side of the edge between their parent and grandparent, which are at an angle of $\frac{2\pi}{\Delta}$ from their parent, as in Figure 2.4.

Once again we normalize (u, v), as having length 1. We then perform a rigid transformation that takes the grandparent, u, to the origin so that the edge (u, v) is along the x-axis, u's closest grandchild, which we call w, is located at the point $\left(1 - f\cos\left(\frac{2\pi}{\Delta}\right), f\sin\left(\frac{2\pi}{\Delta}\right)\right)$. We require that the distance from w to its descendants be no greater than the distance from w to the boundary of the ply disk for u. Recall that our wedge angle $\theta = \frac{2\pi}{\Delta}$. We apply the following constraint:

$$\sqrt{(1 - f\cos\theta)^2 + (f\sin\theta)^2} \geq \frac{\alpha}{f} + \sum_{i=2}^{\infty} f^i$$

After simplifying and solving for α, our condition is

$$\alpha \leq f\sqrt{1 - 2f\cos\theta + f^2} - \frac{f^3}{1 - f}$$

Let us now compare our three conditions. We see that equation 2 gives us an upper bound for f, while equations 1 and 3 give us upper bounds for α that both increase as f gets larger. So to maximize α, we let f be equal to its upper bound. This gives us the following theorem.

Theorem 2.1. *Let T be a tree with maximum degree Δ, and let*

$$f = \frac{\sin\left(\frac{\pi}{\Delta}\right)}{1 + \sin\left(\frac{\pi}{\Delta}\right)}.$$

T has a 1-ply drawing if

$$\alpha \leq \min\left(\frac{f}{1+f},\ f\sqrt{1 - 2f\cos(2\pi/\Delta) + f^2} - \frac{f^3}{1-f}\right).$$

Corollary 2.1. *A tree with maximum degree Δ has a 1 ply drawing when $\alpha = \Theta(1/\Delta)$.*

Proof. First, recall that we defined:

$$f = \frac{\sin\left(\frac{\pi}{\Delta}\right)}{1 + \sin\left(\frac{\pi}{\Delta}\right)}.$$

Now we will consider the limiting value of $\Delta \cdot f$.

$$\lim_{\Delta\to\infty} \Delta \cdot f = \lim_{\Delta\to\infty} \frac{\Delta \sin(\pi/\Delta)}{1 + \sin(\pi/\Delta)} = \pi$$

Therefore, $f = \Theta(1/\Delta)$. So as $\Delta \to \infty$, $f \to 0$.

Secondly, recall that in our theorem we showed:

$$\alpha \leq \min\left(\frac{f}{1+f},\ f\sqrt{1 - 2f\cos(2\pi/\Delta) + f^2} - \frac{f^3}{1-f}\right)$$

Suppose that we use the first condition, $\alpha = f/(1+f)$. Then $\alpha/f = 1/(1+f)$. So $\lim_{f\to 0} \alpha/f = 1$.

Then suppose that we use the second condition:

$$\alpha = f\sqrt{1 - 2f\cos(2\pi/\Delta) + f^2} - \frac{f^3}{1-f}$$

Again, $\lim_{f\to 0} \alpha/f = 1$, so $\alpha = \Theta(f) = \Theta(1/\Delta)$. □

Note, however, that some of our conditions are not tight. For condition 2, we assumed that the branches of our subtrees would approach the sides of their wedge directly. But when the degree of our tree is 4, the angle between two subtrees is 90°. Therefore, every edge in our tree is either horizontal or vertical, so we can measure the distance to the boundary of the wedge using Manhattan distance instead of Euclidean distance. (See Figure 2.5.)

So for a tree with degree 4, we replace condition 2 with the following equation:

$$\sum_{i=1}^{\infty} f^i \leq 1$$

This implies $f = 1/2$, and our other conditions imply $\alpha = 1/3$. In this case, our bound is tight. (See Figure 2.6.)

2.3 Polynomial area, logarithmic ply number

In this section, we prove the following theorem.

Theorem 2.2. *For $\alpha = 0.5$, a tree with maximum degree Δ can be drawn with ply number $O(\log n)$ in area $n^{O(\Delta)}$.*

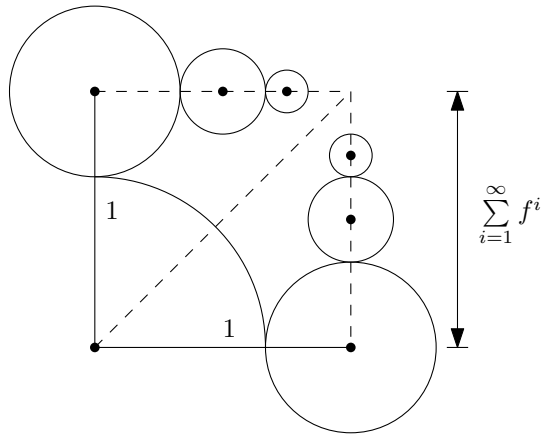

Figure 2.5: An improved bound for Condition 2. The Manhattan distance is sufficient to confine subtrees within a wedge when all edges are either horizontal or vertical.

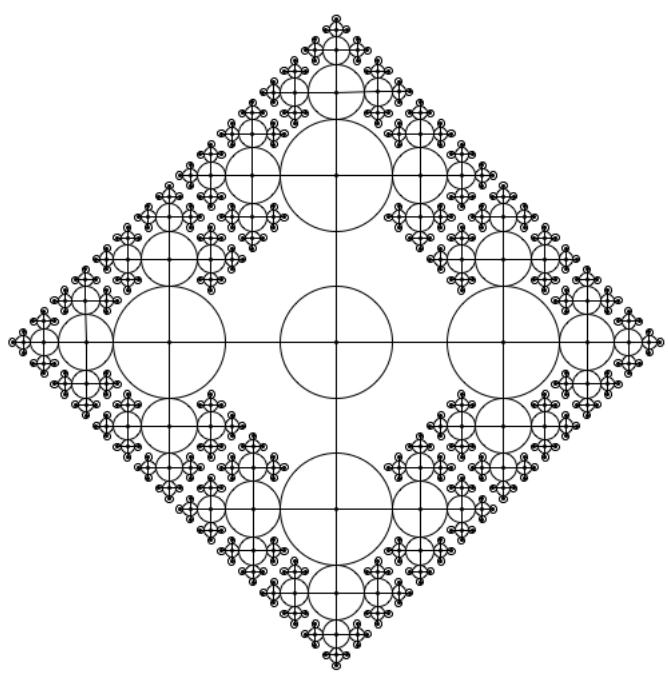

Figure 2.6: A 1-ply drawing of a tree with maximum degree four, for which $f = 1/2$, $\alpha = 1/3$.

Note that for a bounded-degree tree, Δ is a constant, so our area is polynomial in n. We first give a simple fractal layering algorithm that proves our theorem for balanced trees. Then we extend it to all trees by using a heavy path decomposition. A similar approach was used by Angelini *et. al.* [4] for drawing trees up to maximum degree six, but we add our layering technique to make their algorithm work for all trees.

2.3.1 Radially layered drawings

We begin with a simple algorithm for drawing trees by layering their children. For each vertex, we choose a sequence of distances d_i for the layers, such that vertices in adjacent layers have disjoint ply disks.

Lemma 2.1. *Suppose that r is the root of a star graph. Let v_1, v_2 be children at distances d_1, d_2, respectively. If $d_2 \geq 3d_1$, then the ply disks for v_1 and v_2 are disjoint.*

Proof. The distance to v_1 is d_1, so since $\alpha = 0.5$, its ply disk will have radius $0.5d_1$, and will be contained within an open disk of radius $1.5d_1$ centered at r. The distance to v_2 is d_2, so its ply disk will have radius $0.5d_2$. Its closest approach to r will be at distance $0.5d_2 \geq 1.5d_1$. Thus, the ply disks for v_1 and v_2 are disjoint. (See Figure 2.7.) $\square$

Next, note that we can put up to six vertices in each layer without overlaps. So for a tree with degree Δ, we need $\lceil \Delta/6 \rceil$ layers. We pick any desired size for the initial layer around our root, then draw the subtrees for each child vertex recursively within their own ply disks. Therefore, the size of the smallest layer must shrink by a factor of $3^{\lceil \Delta/6 \rceil}$ each time we add a level to our tree.

Since our tree is balanced, its total height is $O(\log n)$. Thus the ratio of the longest to the smallest edge is $3^{O(\Delta \log n)} = n^{O(\Delta)}$. The area will then also be $n^{O(\Delta)}$, for a larger constant.

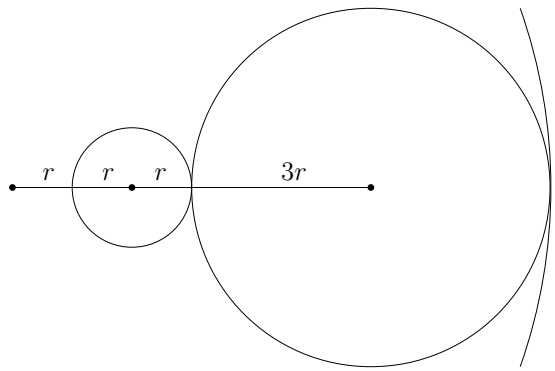

Figure 2.7: If each layer in a tree drawing is at least three times as far as the previous layer, the ply disks for the layers will not overlap. In this figure, $d_1 = 2r$ and $d_2 = 6r$, so our condition holds.

This completes our proof for balanced trees. Figure 2.8 provides an example drawing of such a tree with degree 18 using three layers.

2.3.2 Heavy path decomposition

When our trees are not balanced, we will use the heavy path decomposition [58] to still produce drawings with logarithmic ply number. This decomposition partitions the vertices in our tree into paths that each end at a leaf. To choose the first path, we begin at the root. Then from its child subtrees, we choose the largest one and add its root to our path. We continue downward until we reach a leaf.

We next remove the vertices on this path from our tree, creating a new set of subtrees, and repeat the same process for each subtree. That is, the root vertex for each of these subtrees will become the starting point for a new path constructed by the same process. We recurse until every vertex in our tree is assigned to some path. The subtrees that are rooted at a child of a vertex v and whose root is not on the same path as v are are said to be *anchored* at v. The path containing the root of each of those subtrees is also said to be anchored at v.

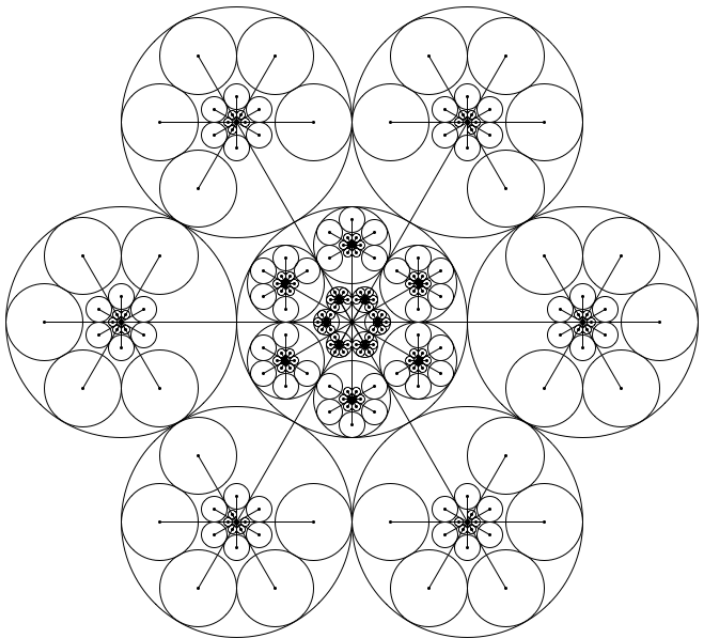

Figure 2.8: A tree with degree 18, where the children of each vertex are drawn in three layers.

The set of paths constructed by this process now itself forms a new tree (see Figure 2.9), in which the path P_i is a parent of P_j if one of the vertices in P_i is an anchor for P_j. We will show that the ply number of our drawings is proportional to the height of this decomposition tree, which is known to be $O(\log n)$.

Now we describe how to draw each path in the decomposition tree. First, we define a *2-drawing* of a path $P = (v_1, \ldots, v_m)$ as a straight-line drawing of P along a single segment that satisfies the following properties.

- All of the vertices appear in the line segment in the same order as they appear in P.
- For each $i = 2, \ldots, m-1$ we have $\frac{l(v_{i-1}, v_i)}{2} \leq l(v_i, v_{i+1}) \leq 2l(v_{i-1}, v_i)$.

Lemma 2.2. *A 2-drawing of a path has ply number at most 2.*

Proof. See Lemma 5 in Angelini *et al.* [4]. □

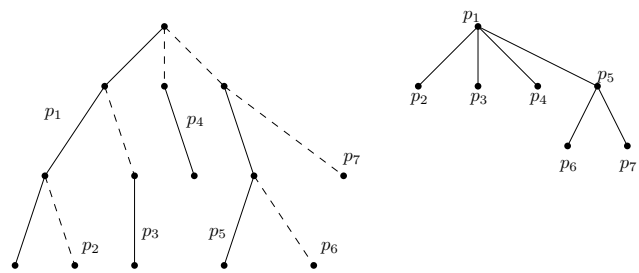

Figure 2.9: A tree and its heavy path decomposition.

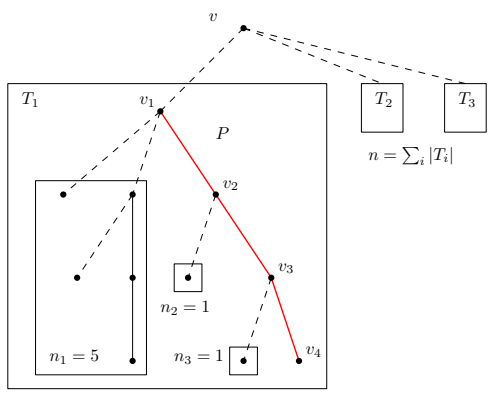

Figure 2.10: Labels for different sizes in a heavy path decomposition tree.

Now suppose that we have a path $P = (v_1, v_2, \ldots, v_k)$ in our heavy path decomposition, and let P be anchored at vertex v, so that v is the parent of v_1. Let n be the total size of the subtrees anchored at v, and let n_i be the total size of the subtrees anchored at v_i (Figure 2.10). Lastly, we denote the length of the edge (v, w) as $l(v, w)$.

Intuitively, we want to draw each path so that more space is available for vertices that have larger subtrees. At the same time, we want to ensure that the lengths of the two edges for a vertex are within a factor of two, so that our path is a 2-drawing. This can be achieved using the following algorithm DrawPath.

To draw the path P, we first set $l(v, v_1) = n_1$ and $l(v_i, v_{i+1}) = n_i + n_{i+1}$, for each $i = 1, \ldots, k-1$. Next we visit the edges of our path in decreasing order of length. When an edge (v_i, v_{i+1}) is visited, we make sure that both of its neighboring edges are at least half as long. That is, we set:

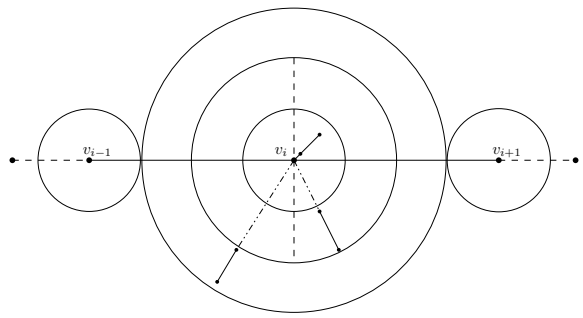

Figure 2.11: Three vertices along a path in our decomposition, along with their drawing disks (not the ply disks). For the center vertex v_i, we show three paths in different layers around it, which would be drawn recursively.

- $l(v_{i-1}, v_i) = \max\{\frac{l(v_i, v_{i+1}}{2}, l(v_{i-1}, v_i)\}$
- $l(v_{i+1}, v_{i+2}) = \max\{\frac{l(v_i, v_{i+1}}{2}, l(v_{i+1}, v_{i+2})\}$

Lemma 2.3. *The algorithm* DrawPath *constructs a 2-drawing* Γ *of* P *such that* $l(v, v_1) \geq n_1$, $l(v_i, v_{i+1}) \geq n_i + n_{i+1}$, *and for each* $i = 1, \ldots, m-1$, *and* $l(P) \leq 6n$.

Proof. See Lemma 6 in Angelini *et al.* [4]. □

We now perform a bottom-up construction of our tree, drawing each path using the DrawPath algorithm. Once all of the paths anchored at vertices in P have been drawn, we construct a drawing of P with each path in a separate layer (Figure 2.11). This translation may increase the ply radius of the first vertex in each of these paths, so the ply number of the drawing for each path may increase from 2 to 3.

We now prove the following properties of this drawing.

Lemma 2.4. *For each vertex* v *we can associate a drawing disk* D_v *(which is distinct from the ply disk for* v*) that satisfies the following properties.*

1. *If* v, w *are two distinct vertices on the same path, then their disks* D_v, D_w *are disjoint.*

2. *The ply disks for the subtrees anchored at v are all contained within D_v, and are within disjoint layers.*

3. *Each path is scaled by a factor of $O(3^{\Delta})$ larger than the paths that are anchored at its vertices.*

Proof. We prove each part of our lemma as follows.

1. Suppose that our heavy path decomposition tree has a total height of H, and the path P is at height h. Then we use the DrawPath algorithm to construct a drawing of P. We set the drawing disk for a vertex v_i in P to have radius n_i, that is, the size of the subtrees anchored at v_i. Since the length of the edge (v_i, v_{i+1}) is at least $n_i + n_{i+1}$ (by Lemma 2.3), the drawing disks for any two adjacent vertices in our path will not overlap.

2. Next we scale the drawing of P by $3^{\Delta(H-h)}$. Note that each path anchored at a vertex in P is scaled by $3^{\Delta(H-(h+1))}$, so the difference in the scaling factor is 3^{Δ}. We show that at least $\Delta - 1$ paths can be anchored in different layers around each vertex v in P.

   From Lemma 2.3, we know that each path anchored at v has an unscaled length of at most $6n$, where n is the total size of the subtrees anchored at v. We also know by Lemma 2.1 that the ply disks for vertices in two different paths will not overlap if their distance from v differs by at least a factor of three.

   So we will draw the jth path anchored at v_i is drawn between x_j and x_{j+1}, where x_j satisfies the following recurrence:

$$x_1 = 6n_i$$

$$x_j = 3x_{j-1} + 6n_i$$

Solving the recurrence, we find that $x_j = 3n_i(3^j - 1)$. Since we have at most $\Delta - 1$ layers, the largest layer will have an outer radius less than $3^\Delta n_i$. Since the unscaled drawing disk for v_i had a radius of n_i, a relative scaling factor of 3^Δ is sufficient to fit the paths that are anchored at it.

3. Since our heavy path decomposition has height $O(\log n)$, the largest path will be scaled by a factor of $3^{O(\Delta \log n)}$ from its original length of $O(n)$. So the diameter of our drawing is $3^{O(\Delta \log n)}n$, which simplifies to $n^{O(\Delta)}$. The total area is then also $n^{O(\Delta)}$, for a larger constant.

□

Together these properties imply that the ply disks for a path can only overlap with ply disks for their ancestor paths in the heavy path decomposition tree. Therefore, since each path is drawn with ply number at most 3, the total ply number is at most $3(h + 1)$, where h is the height of the heavy path decomposition tree. Since $h = O(\log n)$, the ply number is $O(\log n)$.

Lastly, if Δ is a constant, then the total scaling for our largest disk is $3^{O(\Delta \log n)}$, which simplifies to $n^{O(\Delta)}$. This completes our proof of Theorem 2.2.

2.4 Lower bound for 2-trees

Since all trees can be drawn with $O(\log n)$ ply number, it is natural to consider larger planar graph classes. We show that a 2-tree can require at least $\Omega(\sqrt{n/\log n})$ ply, for any fixed α.

First, let us inductively define a 2-tree. As a base case, the graph consisting of two vertices connected by an edge is a 2-tree. Then suppose $G = (V, E)$ is a 2-tree, where V is the set of

vertices $\{v_1, \ldots, v_n\}$, and E is the set of edges. Select an edge $\{v_i, v_j\} \in E$. Then construct $G' = (V', E')$, where $V' = V \cup \{v_{n+1}\}$, and $E' = E \cup \{(v_i, v_{n+1}), (v_j, v_{n+1})\}$. We define G' to also be a 2-tree. Informally, we have constructed G' by taking an edge of G and adding a new vertex connected to both endpoints.

We know that a star can be drawn with ply number 2 when the distance to successive vertices increases exponentially [4]. A tree can be drawn with $O(\log n)$ ply number when the distances from parents to their children decrease exponentially as we move down the tree. Intuitively, combining these two graphs produces a graph that requires large ply, since it is impossible to satisfy both conditions simultaneously.

Accordingly, we begin with m disjoint complete binary trees of height h, which we label T_i, $1 \leq i \leq m$, where m and h will be determined later. Then we add one vertex v connected to every vertex in each tree. Note this graph can be constructed as a subgraph of a 2-tree. A full 2-tree would have additional edges, but we can ignore these for our lower bound since they could only increase the ply number.

Now we have two possible types of drawings for our graph. In one case, every tree has some vertex whose ply disk contains v. Therefore, the ply number of our graph is at least m, since there are at least m ply disks that all contain v. In the second case, there is some tree T_i for which none of the ply disks for its vertices contain v. To analyze this case, we make use of the following lemma.

Lemma 2.5. *If two adjacent vertices w_1 and w_2 satisfy $d(v, w_1) > (1 + 1/\alpha)d(v, w_2)$, then the ply disk for w_2 contains v.*

Proof. Suppose that $d(v, w_2) = r$, so that $d(v, w_1) > (1 + 1/\alpha)r$. By the triangle inequality, $d(w_1, w_2) > r/\alpha$. Then the ply radius for w_1 is at least $\alpha d(w_1, w_2) > r$. Therefore, the ply disk for w_1 contains v. (See Figure 2.12.) $\square$

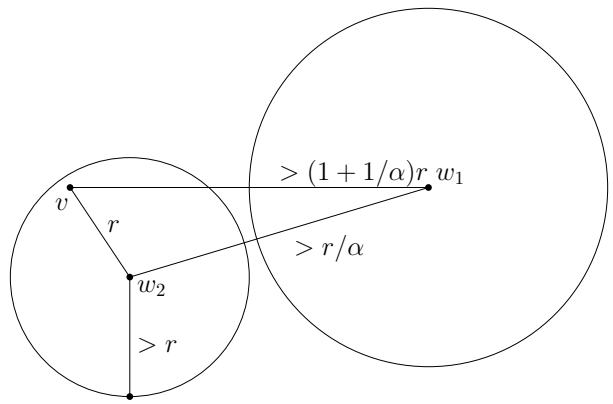

Figure 2.12: If two adjacent vertices w_1 and w_2 are both attached to v, and w_1 is much further away from v, then the ply disk for w_2 will contain v.

Assume without loss of generality that the distance from v to the root of T_i is 1, and let $c = 1 + 1/\alpha$. We can then show by induction that if no ply disk in T_i contains v, then the nodes at the jth level of our tree are at distance at most c^j from v, and at least c^{-j}.

Now partition our drawing into annulae S_l, where the inner radius of S_l is c^l, and the outer radius is c^{l+1}, for $-h \leq l \leq h-1$. Next choose $\bar{l}$ to be the index of the annulus containing the maximum number of vertices. We have more than 2^h vertices, and only $2h$ annulae to distribute our vertices, so $S_{\bar{l}}$ must contain at least $2^h/2h$ vertices. Since each of these vertices is at a distance of at least $c^{\bar{l}}$, each has a ply radius of at least $c^{\bar{l}}\alpha$.

A vertex at a distance of $c^{\bar{l}+1}$ from v will have a ply radius of at least $\alpha c^{\bar{l}+1}$, so the outer edge of its ply disk will be at a distance of $(\alpha+1)c^{\bar{l}+1}$. Therefore, let D be the disk centered at v with a radius of $r_D = (\alpha+1)c^{\bar{l}+1}$, so that all of the ply disks for vertices in $S_{\bar{l}}$ are contained in D. Now we compute the ratio of the areas of the ply disks in D to its own area, which is a lower bound for the ply number. Note that D contains at least $2^h/2h$ ply disks that each have a radius of at least $c^{\bar{l}}\alpha$. Therefore, this ratio is at least:

$$\underbrace{\frac{2^h}{2h}}_{\text{vertices}} \overbrace{(\pi\alpha^2 c^{2\bar{l}})}^{\text{ply area per vertex}} \underbrace{\frac{1}{\pi(\alpha+1)^2 c^{2\bar{l}+2}}}_{\text{inverse disk area}} = \frac{2^h}{h}\frac{\alpha^2}{(\alpha+1)^2 c^2} = \Omega(2^h/h)$$

Now let $h = (\log n + \log\log n)/2$, and let $m = \sqrt{n/\log n}$. Note that the total number of vertices in each tree is $2^{(\log n + \log\log n)/2} = \sqrt{n \log n}$. The total number of vertices overall is then $m \cdot (2^{h+1} - 1) + 1 = O(n)$.

If every tree T_i has a vertex whose ply disk contains v, then the ply number is at least $m = \sqrt{n/\log n}$. Otherwise, if some tree does not have such a vertex, then that tree's ply number is $\Omega(2^h/h) = \Omega(\sqrt{n/\log n})$. This gives us the following theorem.

Theorem 2.3. *There is a 2-tree with $O(n)$ vertices for which any drawing has ply number $\Omega(\sqrt{n/\log n})$, for any fixed $\alpha > 0$.*

2.5 Conclusion

We have shown that all trees have 1-ply drawings when $\alpha = O(1/\Delta)$, or logarithmic ply number when $\alpha = 0.5$, and that 2-trees may require $\Omega(\sqrt{n/\log n})$ ply for any α.

There are many open questions left to resolve, but we are especially interested in closing the gap between constant and logarithmic ply for trees with between three and nine children per node. We would also like to consider intermediate planar graph classes between trees and 2-trees, such as outerplanar graphs, and determine whether they can be drawn with $O(\log n)$ ply.

Chapter 3

Square-Contact Representations of Partial 2-Trees and Triconnected Simply-Nested Graphs *

3.1 Introduction

Contact representations of graphs, in which the vertices of a graph are represented by non-overlapping or non-crossing geometric objects of a specific type, and edges are represented by tangencies or other contacts between these objects, form an important line of research in graph drawing and geometric graph theory. For instance, the Koebe–Andreev–Thurston circle packing theorem states that every planar graph is a contact graph of circles [59]. Other types of contact representations that have been studied include contacts of unit circles [8,36], line segments [41], circular arcs [1], triangles [32], L-shaped polylines [11], and cubes [25].

*Portions of this chapter are included from [14].

Schramm's monster packing theorem [55] implies that every planar graph can be represented by the tangencies of translated and scaled copies of any smooth convex body in the plane. However, it is more difficult to use this theorem for non-smooth shapes, such as polygons: when k bodies can meet at a point, the monster theorem may pack them in a degenerate way in which separating k-cycles, and their interiors, shrink to a single point.

In this chapter we study one of the simplest cases of contact representations that cannot be adequately handled using the monster theorem: contact systems of axis-parallel squares. We distinguish between *proper* and *improper* contacts: a proper contact representation disallows squares that meet only at their corners, while an *improper* or *weak* contact representation allows corner-corner contacts of squares. These weak contacts may represent edges of the graph, but they are also allowed between squares that should be non-adjacent. The weak contact representations by squares were shown by Schramm [56] to include all of the proper induced subgraphs of maximal planar graphs that have no separating 3-cycles or 4-cycles. However, a characterization of the graphs having proper contact representations by squares remains elusive.

There is a simple necessary condition for the existence of a proper contact representation by squares. No three properly-touching squares can surround a nonzero-area region of the plane. Therefore, if every embedding of a planar graph G with four or more vertices has a separating triangle or a triangle as the outer face, then G cannot have a proper contact representation. Our main results show that this necessary condition is also sufficient for two notable families of planar graphs: partial 2-trees (including series-parallel graphs) and triconnected cycle-trees (including the Halin graphs). However, we show that this necessary condition is not sufficient for the existence of weak and proper square-contact representations of 3-outerplanar and 2-outerplanar triconnected simply-nested graphs.

3.2 Preliminaries

A graph is *connected* if it contains a path between any two vertices. A *cutvertex* is a vertex whose removal disconnects the graph. A *separation pair* is a pair of vertices whose removal disconnects the graph. A connected graph is 2-*connected* if it does not have a cutvertex and a 2-connected graph is 3-*connected* if it does not have a separation pair. The maximal 2-connected components of a graphs are its *blocks*.

A graph is *planar* if it admits a drawing in the plane without edge crossings. A *combinatorial embedding* is an equivalence class of planar drawings, where two drawings of a graph are *equivalent* if they determine the same circular orderings for the edges around each vertex. A planar drawing partitions the plane into topologically connected regions, called *faces*. The bounded faces are the *inner faces*, while the unbounded face is the *outer face*. A combinatorial embedding together with a choice for the outer face defines a *planar embedding*. An *embedded graph* (*plane graph*) is a planar graph with a fixed combinatorial embedding (fixed planar embedding).

The graphs considered in this chapter are planar, finite, simple, and connected. We denote the vertex set V and the edge set E of a graph $G = (V, E)$ by $V(G)$ and $E(G)$, respectively. Let H and G be two graphs. We say that G is H-free if G does not contain a subgraph isomorphic to H.

The complete k-partite graph $K_{|V_1|,\ldots,|V_k|}$ is the graph $(V = \bigcup_{i=1}^{k} V_i, E = \bigcup_{i<j} V_i \times V_j)$.

Series-parallel graphs and partial 2-trees. A *two-terminal series-parallel* graph G with source s and target t can be recursively defined as follows: (i) Edge st is a two-terminal series-parallel graph. Let $G_1, \ldots, G_k$ be two-terminal series-parallel graphs and let s_i and t_i be the source and the target of G_i, respectively, with $1 \leq i \leq k$. (ii) The *series composition* of

$G_1, \ldots, G_k$ obtained by identifying s_i with t_{i+1}, for $i = 1, \ldots, k-1$, is a two-terminal series-parallel graph with source s_k and target t_1; and (iii) the *parallel composition* of $G_1, \ldots, G_k$ obtained by identifying s_i with s_1 and t_i with t_1, for $i = 2, \ldots, k$, is a two-terminal series-parallel graph with source s_1 and target t_1.

A *series-parallel graph* is either a single edge or a two-terminal series-parallel graph with the addition of an edge, called *reference edge* joining s and t. Clearly, series-parallel graphs are 2-connected. A series-parallel graph G with reference edge e is naturally associated with a rooted tree T, called the *SPQ*-tree of G. Each internal node of T, with the exception of the one associated with e, corresponds to a two-terminal series-parallel graph. Nodes of T are of three types: *S*-, *P*-, and *Q-nodes*. Further, tree T is rooted to the Q-node corresponding to e.

Let μ be a node of T with terminals s and t and children $\mu_1, \ldots, \mu_k$, if any. Node μ has an associated multigraph, called the *skeleton* of μ and denoted by $skel_\mu$, containing a *virtual edge* $e_i = s_i t_i$, for each child μ_i of μ. Skeleton $skel_\mu$ shows how the children of μ, represented by "virtual edges", are arranged into μ. The skeleton $skel_\mu$ of μ is: (i) edge st, if μ is a leaf Q-node, (ii) the multi-edge obtained by identifying the source s_i and the target t_i of each virtual edge e_i, for $i = 1, \ldots, k$, with a new source s and and new target t, respectively, or (iii) the path $e_1, \ldots, e_k$, where virtual edge e_i and e_{i+1} share vertex $s_i = t_{i+1}$, with $1 \leq i < k$. If μ is an S-node, then we denote by $\ell(\mu)$ the length of $skel_\mu$, i.e., $\ell(\mu) = k$.

For each virtual edge e_i of $skel_\mu$, recursively replace e_i with the skeleton $skel_{\mu_i}$ of its corresponding child μ_i. The two-terminal series-parallel subgraph of G that is obtained in this way is the *pertinent graph* of μ and is denoted by G_μ. We have that G_μ is: (i) edge st, if μ is a Q-node, (ii) the series composition of the two-terminal series-parallel graphs $G_{\mu_1}, \ldots, G_{\mu_k}$, if μ is an S-node, and (iii) the parallel composition of the two-terminal series-parallel graphs $G_{\mu_1}, \ldots, G_{\mu_k}$, if μ is a P-node. We denote by G_μ^- the subgraph of G_μ obtained by removing from it terminals s and t together with their incident edges.

A 2-*tree* is a graph that can be obtained from an edge by repeatedly adding a new vertex connected to two adjacent vertices. Every 2-tree is planar and 2-connected. A *partial* 2-*tree* is a subgraph of a 2-tree. Equivalently, *partial* 2-*trees* can be defined as the K_4-minor-free graphs. In particular, the series-parallel graphs are exactly the 2-connected partial 2-trees.

Simply-nested graphs. Let G be an embedded planar graph and let $G_1, \ldots, G_k$ be the sequence of embedded planar graphs such that $G_1 = G$, graph G_{i+1} is obtained from G_i be removing all the vertices incident to the outer face of G_i together with their incident edges, and G_k is outerplanar. We say that the embedding of G is *k-outerplanar*. A graph is *k-outerplanar* if it admits a k-outerplanar embedding. The set V_i of vertices incident to the outer face of G_i is the *i-th level* of G. A k-outerplanar graph is *simply-nested* [12] if, for $i = 1, \ldots, k-1$, graphs $G[V_i]$ are chordless cycles and $G[V_k]$ is either a cycle or a tree.

We define *cycle-trees* and *cycle-cycles* the 2-outerplanar simply-nested graphs whose internal level is a tree and a cycle, respectively. The 2-outerplanar 3-connected simply-nested graphs have a nice geometric interpretation. Similarly to the Halin graphs, which are the graphs of polyhedra containing a face that share an edge with all other faces, 3-connected cycle-trees are the graphs of polyhedra containing a face touched by all other faces. Analogously, the 3-connected cycle-cycle graphs with no chords on the inner cycle are the graphs of polyhedra in which there exist two disjoint faces that are both touched by all other faces.

Square-contact representations. Let $G = (V, E)$ be a planar graph. A *square-contact representation* Γ of G maps each vertex $v \in V$ to an axis-aligned square $S_\Gamma(v)$ in the plane, such that, for any two vertices $u, v \in V$, squares $S_\Gamma(u)$ and $S_\Gamma(v)$ are interior-disjoint, and the sides of $S_\Gamma(u)$ and $S_\Gamma(v)$ touch if and only if $uv \in E$. A square-contact representation of G is *proper* if any two touching squares share infinitely many points, i.e., they cannot share only a corner point, and *non-proper*, otherwise. When the square-contact representation is clear

from the context, we may choose to drop the Γ subscript and just use $S(v)$ to refer to the square for vertex v. In the remainder of the chapter, we only consider proper square-contact representations and refer to such representations simply as square-contact representations.

Geometric transformations. Let G be planar graph and let Γ be a square-contact representation of G. Also, let p be any point in Γ. We define the $\nearrow$-, $\nwarrow$-, $\swarrow$-, and $\searrow$-*quadrant* of p in Γ as the first, second, third, and fourth quadrant around p, respectively. Suppose that the half-lines delimiting the $\swarrow$-quadrant of p in Γ do not intersect the interior of any square in Γ. Also, let Γ' be the part of Γ lying in the $\swarrow$-quadrant of p. Then, a $\overset{\swarrow}{p}$*-scaling* of Γ *by a factor* $\alpha > 0$ is a square-contact representation Γ^* defined as follows; see, e.g., Figure 3.3. Initialize $\Gamma^* = \Gamma$ and remove from Γ^* the drawing of the squares contained in the interior of Γ'. Then, insert into Γ^* a copy Γ'' of Γ' scaled by α such that the upper-right corner of Γ'' coincides with p. Clearly, depending on the scale factor α, drawing Γ^* may or may not be a square-contact representation of G (as adjacencies may be lost or gained). In the following, we refer to the case in which $\alpha > 1$ simply as a $\overset{\swarrow}{p}$*-scaling* of Γ and to the case in which $0 < \alpha < 1$ as a *negative* $\overset{\swarrow}{p}$*-scaling* of Γ. The definitions of $\overset{\circ}{p}$*-scaling* and *negative* $\overset{\circ}{p}$*-scaling*, with $\circ \in \{\nwarrow, \searrow, \nearrow\}$, are analogous. Finally, let v be a vertex of G and let x, y, z, and w be the upper-left, lower-left, lower-right, and upper-left corner points of $S(v)$ in Γ. A $\overset{\nwarrow}{v}$*-scaling*, $\overset{\swarrow}{v}$*-scaling*, $\overset{\searrow}{v}$*-scaling*, $\overset{\nearrow}{v}$*-scaling* of Γ is a $\overset{\nwarrow}{x}$*-scaling*, $\overset{\swarrow}{y}$*-scaling*, $\overset{\searrow}{z}$*-scaling*, $\overset{\nearrow}{w}$*-scaling* of Γ, respectively.

3.3 Partial 2-Trees

In this section, we study square-contact representations of partial 2-trees and give the following simple characterization for graphs in this family admitting such representations.

Theorem 3.1. *Let G be a partial 2-tree. Then, the following statements are equivalent:*

(i) G is $K_{1,1,3}$-free,

(ii) G admits an embedding without separating triangles, and

(iii) G admits a square-contact representation.

In order to prove THEOREM 3.1, we first show that, without loss of generality, we can restrict our attention to the biconnected partial 2-trees, i.e., the series-parallel graphs.

LEMMA 3.1. *Let G be a $K_{1,1,3}$-free partial 2-tree. Then, there exists a $K_{1,1,3}$-free series-parallel graph G^* such that $G \subset G^*$ and G admits a square-contact representation if G^* does.*

Proof. Let $\beta(H)$ denote the number of blocks of a graph H. We show that G can be augmented to a $K_{1,1,3}$-free partial 2-tree G' such that $\beta(G') = \beta(G) - 1$, by adding to G a new vertex connected to two vertices in $V(G)$ incident to the same cut-vertex of G, belonging to different blocks, and sharing a common face. Hence, repeating such an augmentation eventually yields a series-parallel graph G^* that is $K_{1,1,3}$-free. Also, by construction, two vertices in $V(G)$ are adjacent in G^* if and only if they are adjacent in G. Therefore, a square-contact representation of G can be derived from a square-contact representation Γ^* of G^*, by removing from Γ^* all the squares corresponding to vertices in $V(G^*) \setminus V(G)$.

Suppose that $\beta(G) > 1$, as otherwise G is 2-connected and we can set $G^* = G$. Consider a planar embedding $\mathcal{E}$ of G and a cut-vertex c of G. Let $e_1 = (c, u)$ and $e_2 = (c, v)$ be two edges incident to c such that (i) e_1 and e_2 belong to distinct blocks B_1 and B_2 of G, respectively, and (ii) e_1 precedes e_2 in the clockwise order of the edges incident to c in $\mathcal{E}$. Let f be the face of $\mathcal{E}$ that lies to the right of e_1 (to the left of e_2) when traversing e_1 (e_2) from c to u (from c to v). Augment G to graph G' by adding a new vertex w and edges $e'_1 = (w, u)$ and $e'_2 = (w, v)$ inside f. Clearly, blocks B_1 and B_2 of G are now "merged" in a single block B in G'. Also, G' is a partial 2-tree. In fact, since G is a partial 2-teee and since any path connecting a vertex in $V(B_1)$ and a vertex in $V(B_2)$ must pass either through c or trough w,

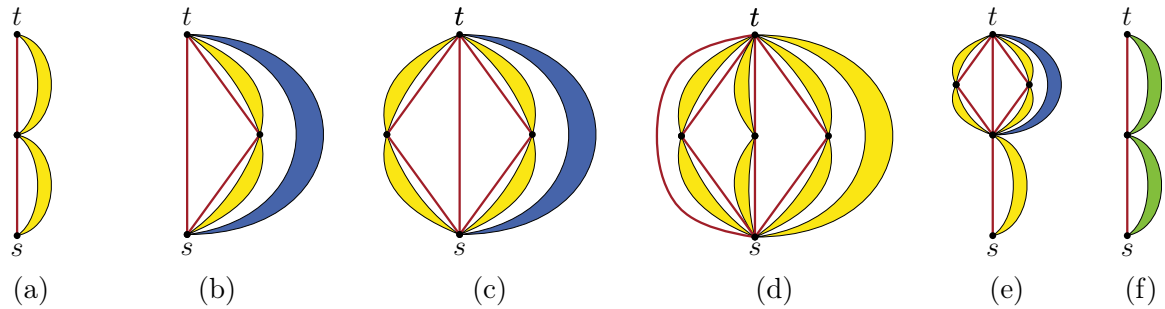

Figure 3.1: (a) A critical S-node, (b) an almost-bad P-node, (c) a bad P-node, (d) a forbidden P-node, (e) an S-node of `Type B`, and (f) an S-node of `Type C`. Yellow, green, and blue regions represent parallel compositions of any number of S-nodes, at most one critical S-node and any number of non-critical S-nodes, and any number of non-critical S-nodes, respectively.

graph G' does not contain K_4 as a minor. Finally, G' is $K_{1,1,3}$-free. This is due to the fact that G is $K_{1,1,3}$-free and that, since u and v belong to different blocks of G, these vertices do not belong to an induced $K_{1,1,2}$ subgraph of G. □

As already observed in section 3.1, an embedding without separating triangles is necessary for the existence of a square-contact representation, and $K_{1,1,3}$ has no embedding without separating triangles. Thus, $(iii) \Rightarrow (ii) \Rightarrow (i)$ are immediate. To complete the proof of Theorem 3.1, we show how to construct a square-contact representation of any $K_{1,1,3}$-free series-parallel graph, proving that $(i) \Rightarrow (iii)$. We formalize this result in the next theorem.

Theorem 3.2. *Every $K_{1,1,3}$-free series-parallel graph admits a square-contact representation.*

Let G be a series-parallel graph and let T be the SPQ-tree of G with respect to any reference edge. We start with some definitions; refer to Figure 3.1. Let μ be an S-node in T. We say that μ is *critical*, if $skel_\mu = s$–x–t and the two children of μ both contain an edge between their terminals, i.e., $sx, xt \in E(G_\mu)$, and *non-critical*, otherwise. Let μ be a P-node in T containing an edge between its terminals. We say that μ is *almost bad*, if it has exactly one critical child, *bad*, if it has exactly two critical children, and *forbidden*, if it has more than

two critical children. Finally, let μ be a P-node in T. We say that μ is *good*, if it is neither bad, nor almost bad, nor forbidden.

We now assign one of three possible types to each S-node μ in T as follows (for each child μ_i of μ, we denote the two terminals of G_{μ_i} as s_i and t_i).

Type A Node μ is of `Type A`, if either $\ell(\mu) > 2$ or $\ell(\mu) = 2$ and at least one child of μ does not contain an edge between its terminals, i.e., $|\{s_1t_1, s_2t_2\} \cap E(G_\mu)| < 2$.

Type B Node μ is of `Type B`, if $\ell(\mu) = 2$, all its children contain an edge between their terminals, and at least one of them is a bad P-node.

Type C Node μ is of `Type C`, if $\ell(\mu) = 2$, and all its children contain an edge between their terminals, and none of them is a bad P-node.

Observe that S-nodes of `Type B` and of `Type C` are also critical.

Let G be a $K_{1,1,3}$-free series-parallel graph and let T be the SPQ-tree of G with respect to any reference edge. We have the following simple observations regarding the P-nodes in T.

Observation 1. *SPQ-tree T contains no forbidden P-node; refer to Figure 3.1(d).*

Observation 2. *Let μ be a P-node in T with terminals s and t such that $st \in E(G_\mu)$. Then, none of the children of μ is of `Type B` and at most two children of μ are of `Type C`.*

We now consider special square-contact representations for the pertinent graphs of the S-nodes in T. Let Γ_μ be a square-contact representation of G_μ. We say that Γ_μ is either a *rectangular*, *L-shape*, or *pipe drawing* of G_μ, if it satisfies the following conditions; refer to Figure 3.2.

Rectangular drawing $S(t)$ lies to the left and above $S(s)$ and the drawing Γ_μ^- of G_μ^- in Γ_μ lies to the right of $S(t)$ and above $S(s)$; also, all the squares of Γ_μ^- whose left side

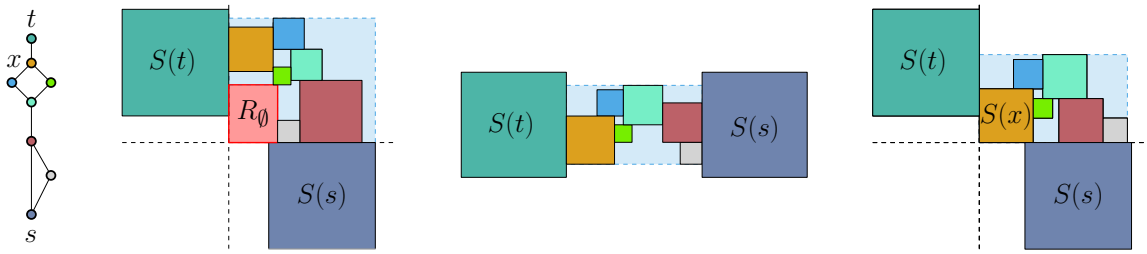

Figure 3.2: From left to right: pertinent G_μ of an S-node μ with terminals s and t, L-shape and pipe drawings of G_μ, respectively, and a rectangular drawing of an S-node ν with pertinent $G_\nu = G_\mu \cup sx$. The L-shape region and horizontal pipe enclosing G_μ^- and the rectangle enclosing G_ν^- are shaded blue.

(bottom side) is collinear with the right side of $S(t)$ (with the top side of $S(s)$) are adjacent to $S(t)$ (to $S(s)$).

L-shape drawing Γ_μ is a rectangular drawing in which there exists a rectangular region (red region $R_\emptyset$ in Figure 3.2) inside the bounding box of Γ_μ^- whose interior does not intersect any square in Γ_μ^- and whose lower-left corner lies at the intersection point between the vertical line passing through the right side of $S(t)$ and the horizontal line passing through the top side of $S(s)$.

Pipe drawing $S(t)$ lies to the left of $S(s)$ and the drawing Γ_μ^- of G_μ^- in Γ_μ lies to the right of $S(t)$ and to the left of $S(s)$; also, all the squares of Γ_μ^- whose left side (right side) is collinear with the right side of $S(t)$ (with the left side of $S(s)$) are adjacent to $S(t)$ (to $S(s)$).

In the following, we generally refer to a drawing of an S-node μ in T (of G_μ) which is either an L-shape drawing, a pipe drawing, or a rectangular drawing as a *valid drawing* of μ (of G_μ).

Let Γ_μ^- be the square-contact representation of G_μ^- contained in Γ_μ. Observe that Γ_μ^- lies in the interior of an orthogonal hexagon with an internal angle equal to 270°, i.e., an *L-shape polygon* (or, simply, *L-shape*), if Γ_μ^- is an L-shape drawing. Also, Γ_μ^- lies in the interior of a rectangle whose opposite vertical sides are adjacent to the right side of $S(t)$ and to the left side of $S(s)$, i.e., a *horizontal pipe*, if Γ_μ^- is a pipe drawing. Finally, Γ_μ^- lies in the interior of a

rectangle whose left and bottom side are adjacent to the right side of $S(t)$ and to the top side of $S(s)$, respectively, if Γ_μ^- is a rectangular drawing.

Proof of Theorem 3.2. In order to prove Theorem 3.2, we proceed as follows. Let G be a $K_{1,1,3}$-free series-parallel graph and let T be the SPQ-tree of G rooted at a Q-node ρ with terminals s and t, whose unique child τ is an S-node. Observe that such a Q-node always exists, since G is simple, and that node τ is either of `Type A` or of `Type C`, since G is $K_{1,1,3}$-tree. We perform a bottom-up traversal in T to construct one or two valid drawings of G_μ, for each S-node $\mu \in T$. Namely, we compute:

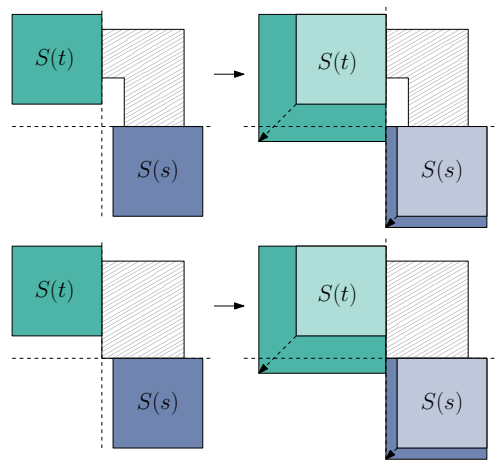

Figure 3.3: Transforming Γ_τ into Γ_ρ.

- an L-shape drawing, if μ is of `Type A` (Lemma 3.4),
- a pipe drawing, if μ is of `Type B` (Lemma 3.5), and
- both a pipe drawing and a rectangular drawing, if μ is of `Type C` (Lemma 3.6).

Thus, when node τ is considered, we can compute either an L-shape drawing of G_τ, if τ is of `Type A`, or a rectangular drawing of G_τ, if τ is of `Type C`. Further, both such valid drawings Γ_τ of G_τ can be easily turned into a square-contact representation Γ_ρ of $G = G_\tau \cup st$, by performing a $\overset{\swarrow}{t}$-scaling and an $\overset{\swarrow}{s}$-scaling of Γ_τ in such a way that the right side of $S(t)$ and the left side of $S(s)$ touch; refer to Figure 3.3. This is possible since both in an L-shape drawing and in a rectangular drawing of G_τ all the squares of G_τ^- whose left side (bottom side) is collinear with the right side of $S(t)$ (with the top side of $S(s)$) are adjacent to $S(t)$ (to $S(s)$).

Let μ be an S-node and let $\mu_1, \ldots, \mu_k$ be the children of μ in T. If each child μ_i of μ is a Q-node, then node μ is of `Type A`, if $\ell(\mu) > 2$, and it is of `Type C`, otherwise. It is not difficult to see that, in the former case, G_μ admits an L-shape drawing and that, in the latter case, G_μ admits both a pipe drawing and a rectangular drawing. In the remainder of the section, we consider the case in which μ has both Q-node and P-node children.

We first show how to construct special square-contact representations of G_μ, that we call *canonical drawings*, for any P-node μ in T, assuming that valid drawings have been computed for each S-node child of μ. We distinguish five possible canonical drawings, depending on 1. the number and type of the S-node children of μ and 2. the presence of edge st. Each canonical drawing has three variants: `vertical` (V), `horizontal` (H), and `diagonal` (D). We name such canonical representations XY *drawings*, where $X \in \{V, H, D\}$ denotes the variant of the representation and $Y = 1$, if $st \in E(G_\mu)$, and $Y = 0$, otherwise. Canonical drawings share the following main property (which, in fact, also holds for valid drawings).

Property 1. *Let Γ_μ be a valid drawing or a canonical drawing of G_μ. Then, for each vertex v in $V(G_\mu^-)$, it holds that $vs \in E(G_\mu)$ ($vt \in E(G_\mu)$) if:*

1. *$S(v)$ has a side that is collinear with a side of $S(s)$ (of $S(t)$) in Γ_μ and*
2. *$S(v)$ is separated from $S(s)$ (from $S(t)$) in Γ_μ by the line passing through such a side.*

Property 1 allows us to modify canonical and valid drawings by appropriate $\overset{\circ}{s}$-scaling and $\overset{\circ}{t}$-scaling transformations, with $\circ \in \{\nwarrow, \nearrow, \searrow, \swarrow\}$, preserving adjacencies between vertices in G_μ.

First, consider a P-node μ in T with terminals s and t such that $st \notin E(G_\mu)$ and let $\mu_1, \ldots, \mu_k$ be the S-node children of μ. We say that a square-contact representation Γ_μ of G_μ is an *H0 drawing* or a *V0 drawing*, if it satisfies the following conditions (in addition to **Property** 1); refer to Figure 3.4.

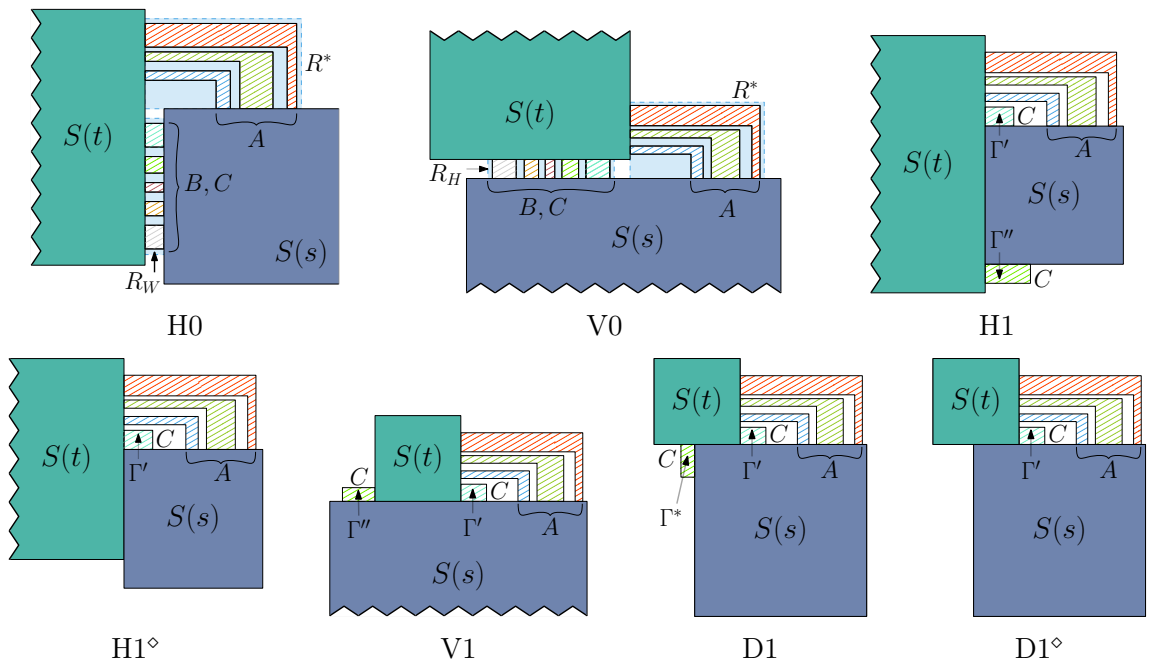

Figure 3.4: Canonical drawings of a P-node μ. The striped regions correspond to L-shapes, horizontal pipes, and rectangles enclosing the square-contact representations of graphs $G^-_{\mu_i}$, for each S-node child μ_i of μ. Labels A, B, and C indicate the type of each S-node.

H0 drawing $S(t)$ lies to the left of $S(s)$, the bottom side of $S(s)$ lies below the bottom side of $S(t)$, and the drawing of G^-_μ in Γ_μ lies to the right of $S(t)$, below the top side of $S(t)$, above the bottom side of $S(s)$, and to the left of the right side of $S(s)$.

V0 drawing $S(t)$ lies above $S(s)$, the left side of $S(s)$ lies to the right of the left side of $S(t)$, and the drawing of G^-_μ in Γ_μ lies above $S(s)$, to the right of the left side of $S(s)$, below the top side of $S(t)$, and to the left of the right side of $S(s)$.

Now, consider a P-node μ in T with terminals s and t such that $st \in E(G_\mu)$ and let $\mu_1, \ldots, \mu_k$ be the S-node children of μ. We say that a square-contact representation Γ_μ of G_μ is an *H1 drawing*, an *H1$^\diamond$ drawing*, a *V1 drawing*, a *D1 drawing*, or a *D1$^\diamond$ drawing*, if it satisfies the following conditions (in addition to **Property** 1); refer to Figure 3.4.

H1 drawing $S(t)$ lies to the left of $S(s)$, the bottom side of $S(s)$ lies above the bottom side of $S(t)$, and the drawing of G_μ^- in Γ_μ lies to the right of $S(t)$, below the top side of $S(t)$, above the bottom side of $S(t)$, and to the left of the right side of $S(s)$.

H1$^\diamond$ drawing $S(t)$ lies to the left of $S(s)$, the bottom side of $S(s)$ lies below the bottom side of $S(t)$, and the drawing of G_μ^- in Γ_μ lies to the right of $S(t)$, below the top side of $S(t)$, above the top side of $S(s)$, and to the left of the right side of $S(s)$.

V1 drawing $S(t)$ lies above $S(s)$ and the drawing of G_μ^- in Γ_μ lies above $S(s)$, below the top side of $S(t)$, to the right of the left side of $S(s)$, and to the left of the right side of $S(s)$.

D1 drawing $S(t)$ lies above $S(s)$ and the left side of $S(t)$ lies to the left of the left side of $S(s)$, and the drawing of G_μ^- in Γ_μ lies to the right of the left side of $S(t)$, below the top side of $S(t)$, above the bottom side of $S(s)$, and to the left of the right side of $S(s)$.

D1$^\diamond$ drawing Γ_μ is a D1 drawing of G_μ in which the drawing of G_μ^- lies to the right of $S(t)$.

We now present two lemmata for the possible canonical drawings of each P-node μ in T. Recall that, by Observation 1, we can assume that μ is not a forbidden P-node. Let $\mu_1, \ldots, \mu_k$ be the S-node children of μ. The general strategy in the proofs of both lemmata consists of 1. computing appropriate valid drawings $\Gamma_{\mu_1}, \ldots, \Gamma_{\mu_k}$ for the pertinent graphs $G_{\mu_1}, \ldots, G_{\mu_k}$ of $\mu_1, \ldots, \mu_k$, respectively, 2. modifying the square-contact representation of $G_{\mu_i}^-$ contained in Γ_{μ_i}, for $i = 1, \ldots, k$, by means of affine transformations, so that representations derived from S-nodes of the same type lie in the interior of the same polygon, and finally 3. composing the resulting drawings into a canonical drawing of G_μ.

We first consider the case in which μ does not contain an edge between its terminals. In this case, by Lemmata 3.4, 3.5, and 3.6, we can assume that Γ_{μ_i} is an L-shape drawing, if μ_i is of `Type A`, and a pipe drawing, if μ_i is of `Type B` or of `Type C`, for $i = 1, \ldots, k$.

Lemma 3.2. *Let μ be a P-node in T with terminals s and t such that $st \notin E(G_\mu)$. Then, graph G_μ admits an H0 drawing and a V0 drawing.*

Proof. In order to obtain an H0 drawing Γ_{H0} of G_μ (a V0 drawing Γ_{V0} of G_μ) we compose the drawings Γ_{μ_i} of G_{μ_i}, with $1 \leq i \leq k$, as depicted in Figure 3.4(H0) (in Figure 3.4(V0)). Specifically, we proceed as follows. First, we scale all the pipe drawings of the children of μ of `Type B` and of `Type C` in such a way that they have the same width W (the same height H). Then, we compose these drawings in such a way for them to fit in a rectangle R_W of width W (a rectangle R_H of height H) and such that no two pipes overlap; let Γ_P be the resulting drawing. Similarly, we scale all the L-shape drawings of the children of μ of `Type A` in such a way that they fit into a rectangle R^*, the left side of each L-shape is incident to the left side of R^*, the bottom side of each L-shape is incident to the bottom side of R^*, and no two L-shapes overlap; let Γ_L be the resulting drawing. Then, we draw u and v as squares $S(u)$ and $S(v)$ of appropriate size in Γ_{H0} (in Γ_{V0}) in such a way that $S(t)$ is incident to the left side of R_W (top side of R_H), $S(s)$ is incident to the right side of R_W (bottom side of R_H). Finally, we insert a scaled copy of Γ_L in Γ_{H0} (in Γ_{V0}) such that the left side of each L-shape contained in Γ_L is adjacent to the right side of $S(t)$ and the bottom side of each L-shape contained in Γ_L is adjacent to the top side of $S(s)$. This concludes the construction of Γ_{H0} and of Γ_{V0}. □

Then, we consider the case in which μ contains an edge between its terminals. Recall that, by Observation 2, node μ has no child of `Type B` and at most two children of `Type C`. In particular, node μ has two children of `Type C`, if it is bad, and one child of `Type C`, if it is almost bad. In this case, by Lemmata 3.4 and 3.6, we can assume that Γ_{μ_i} is an L-shape drawing, if μ_i is of `Type A`, and a rectangular drawing, if μ_i is of `Type C`, for $i = 1, \ldots, k$.

Lemma 3.3. *Let μ be a P-node in T with terminals s and t such that $st \in E(G_\mu)$. Then, graph G_μ admits*

- *an H1 drawing, a V1 drawing, and a D1 drawing, if μ is bad, or*
- *an H1$^\diamond$ drawing and a D1$^\diamond$ drawing, if μ is good or almost bad.*

Proof. We define rectangle R^* and the drawing Γ_L enclosing the drawing of $G^-_{\mu_i}$ in the L-shape drawing Γ_{μ_i} of each child μ_i of μ of `Type A` exactly as in the proof of LEMMA 3.2.

We prove the first part of the statement. Let μ_1 and μ_2 be the two S-node children of μ of `Type C` and let Γ_{μ_1} and Γ_{μ_1} be rectangular drawings of G_{μ_1} and G_{μ_2}, respectively. We show how to construct an H1 drawing Γ_{H1} of G_μ. The construction of a V1 drawing being symmetric. Refer to Figs. 3.4(H1) and 3.4(V1). First, we draw u and v as squares $S(u)$ and $S(v)$ of appropriate size such that $S(t)$ lies to the left of $S(s)$, the bottom side of $S(s)$ lies above the bottom side of $S(t)$, the top side of $S(s)$ lies below the top side of $S(t)$, and the right side of $S(t)$ is adjacent to the left side of $S(s)$; let Γ_H be the resulting drawing. Then, we insert a scaled copy of Γ_L in Γ_H such that the left side of each L-shape contained in Γ_L is adjacent to the right side of $S(t)$ and the bottom side of each L-shape contained in Γ_L is adjacent to the top side of $S(s)$. We obtain Γ_{H1} from Γ_H as follows. First, we insert a scaled copy Γ' of $G^-_{\mu_1}$ in Γ_{μ_1} in the interior of Γ_L in such a way that the left side and the bottom side of the rectangle enclosing Γ' is adjacent to the right side of $S(t)$ and to the top side of $S(s)$, respectively, and Γ' does not overlap with any square in Γ_L. Finally, consider the drawing Γ'' of $G^-_{\mu_2}$ in Γ_{μ_2} after being mirrored with respect to the x-axis. We insert a scaled copy of Γ'' in Γ_H in such a way that the top side and the left side of the rectangle enclosing Γ'' is adjacent to the right side of $S(s)$ and to the bottom side of $S(t)$, respectively. Observe that the flip of the drawing of $G^-_{\mu_2}$ with respect to the x-axis has been performed to preserve adjacencies in the final drawing, as we are placing Γ'' below $S(s)$. We now show how to construct a D1 drawing Γ_{D1} of G_μ. Refer to Figure 3.4(D1). First, we draw u and v as squares $S(u)$ and $S(v)$ of appropriate size such that $S(t)$ lies above $S(s)$, the left side of $S(t)$ lies to the left of the left side of $S(s)$, the right side of $S(t)$ lies between the left and

the right side of $S(s)$, and the bottom side of $S(t)$ is adjacent to the top side of $S(s)$; let Γ_D be the resulting drawing. Then, we extend Γ_D by inserting in it a scaled copy of Γ_L and a scaled copy Γ' of $G^-_{\mu_1}$ in Γ_{μ_1} as discussed above for constructing an H1 drawing. Observe that Γ_D is now a D1$^\diamond$ drawing. Finally, consider the drawing Γ^* of $G^-_{\mu_2}$ in Γ_{μ_2} after being mirrored with respect to the y-axis and rotated by $-90°$. We insert a scaled copy of Γ^* in Γ_D in such a way that the top side and the right side of the rectangle enclosing Γ^* is adjacent to the bottom side of $S(t)$ and to the left side of $S(s)$, respectively. Observe that the flip of the drawing of $G^-_{\mu_2}$ with respect to the y-axis and the counter-clockwise rotation by $90°$ have been performed to preserve adjacencies in the final drawing, as we are placing Γ^* below $S(t)$ and to the left of $S(s)$.

Now, we prove the second part of the statement. Observe that, since μ is good or almost bad, it has at most one S-node child of `Type C`, as S-nodes of `Type C` are critical. For the construction of an H1$^\diamond$ drawing and of a D1$^\diamond$ drawing we proceed as discussed above for the construction of Γ_{H1} and of Γ_{D1}, respectively. However, in this case, we can simply omit the last step in these constructions, in which we extend Γ_{H1} and Γ_{D1} with drawings Γ'' and Γ^*, respectively. As observed above, the construction of Γ_{D1} immediately yields a D1$^\diamond$ drawing, if μ is good or almost bad. Refer to Figure 3.4(D1$^\diamond$). Instead, in order to obtain an H1$^\diamond$ drawing from Γ_{H1}, we only need to perform a final negative $\acute{t}$-scaling of Γ_{H1} so that the bottom side of $S(t)$ lies above the bottom side of $S(s)$. Refer to Figure 3.4(H1$^\diamond$). This concludes the proof. □

We finally turn our attention to the valid drawings of the S-nodes in T. Let μ be an S-node in T and let $\mu_1, \ldots, \mu_k$ be the children of μ (where the virtual edge e_i, corresponding to node μ_i, precedes the virtual edge e_{i+1}, corresponding to node μ_{i+1}, from t to s in $skel_\mu$). The next three lemmata immediately imply Theorem 3.2. To simplify their proofs, we assume that each child of μ is a P-node. In fact, the case in which a child of μ is a Q-node can be treated analogously to that of a P-node containing an edge between its terminals.

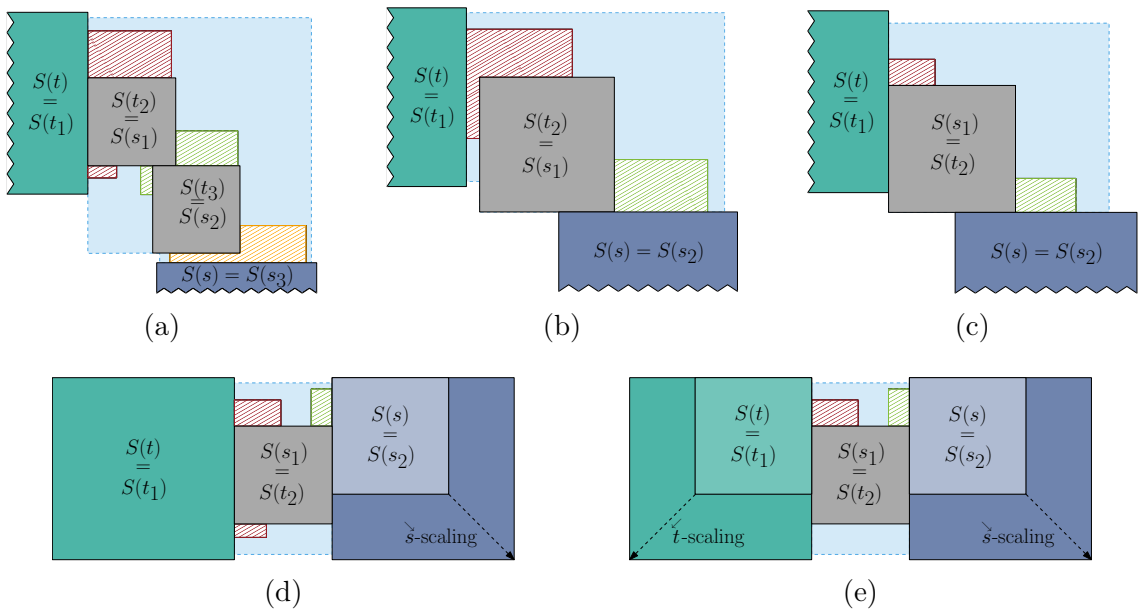

Figure 3.5: Illustrations for the proofs of Lemmata 3.4, 3.5, and 3.6. Striped polygons of the same color enclose different parts of the drawing of each graph $G^-_{\mu_i}$ (contained in the canonical drawing Γ_{μ_i} of G_{μ_i}). (a) An H1 drawing of G_{μ_1}, a D1 drawing of G_{μ_2}, and a D1$^\diamond$ drawing of G_{μ_3} are combined into an L-shape drawing. (b) An H0 drawing of G_{μ_1} and a D1$^\diamond$ drawing of G_{μ_2} are combined into an L-shape drawing. (c) An H1$^\diamond$ drawing of G_{μ_1} and a D1$^\diamond$ drawing of G_{μ_2} are combined into a rectangular drawing. (d) An H1 drawing of G_{μ_1} and a D1$^\diamond$ drawing of G_{μ_2} are combined into a pipe drawing. (e) An H1$^\diamond$ drawing of G_{μ_1} and a D1$^\diamond$ drawing of G_{μ_2} are combined into a pipe drawing.

The general strategy in the proofs of all three lemmata consists of 1. computing appropriate canonical drawings $\Gamma_{\mu_1}, \ldots, \Gamma_{\mu_k}$ for the pertinent graphs $G_{\mu_1}, \ldots, G_{\mu_k}$ of $\mu_1, \ldots, \mu_k$, respectively, 2. modifying these drawings, by means of affine transformations, so that the squares corresponding to terminals shared by different children of μ can be identified without introducing any overlapping between squares corresponding to internal vertices of G_{μ_i} and G_{μ_j}, with $i \neq j$, and finally 3. composing the resulting drawings into a valid drawing of G_μ.

Lemma 3.4. *If μ is an S-node of* `Type A`*, then G_μ admits an L-shape drawing.*

Proof. We first describe how to select a valid drawing of Γ_{μ_i} of G_{μ_i}, for $i = 1, \ldots, k$, based on whether (i) $\ell(\mu) > 2$ or (ii) $\ell(\mu) = 2$. Recall that, if $\ell(\mu) = 2$, then at least one child of μ does not contain an edge between its terminals, say μ_1 (the case in which $s_1t_1 \in E(G_{\mu_1})$ and $s_2t_2 \notin E(G_{\mu_2})$ is analogous).

(i) By Lemma 3.2 and Lemma 3.3, we can construct a drawing Γ_{μ_i}, for each μ_i, such that: (a) Γ_{μ_1} is an H0 drawing, if $s_1t_1 \notin E(G_{\mu_1})$, and Γ_{μ_1} is an H1 drawing (H1$^\diamond$ drawing), if μ_1 is bad (if μ_1 is good or almost bad); (b) Γ_{μ_2} is a V0 drawing, if $s_2t_2 \notin E(G_{\mu_2})$, and Γ_{μ_2} is a D1 drawing (D1$^\diamond$ drawing), if μ_2 is bad (if μ_2 is good or almost bad); and (c) Γ_{μ_i} is a V0 drawing, if $s_it_i \notin E(G_{\mu_i})$, and Γ_{μ_i} is a V1 drawing (D1$^\diamond$ drawing), if μ_i is bad (if μ_i is good or almost bad), for every $i > 2$.

(ii) By Lemma 3.2 and Lemma 3.3, we can construct an H0 drawing Γ_{μ_1} of G_{μ_1} and a V1 drawing (D1$^\diamond$ drawing) Γ_{μ_2} of G_{μ_2}, if μ_2 is bad (if μ_2 is good or almost bad).

We show how to compose all such drawings into an L-shape drawing Γ_μ of G_μ as follows. Refer to Figure 3.5(a) for an example of how to compose drawings Γ_{μ_i}, with $i = 1, \ldots, k$, in case (i) and to Figure 3.5(b) for an example of how to compose drawings Γ_{μ_1} and Γ_{μ_2} in case (ii). First, we scale $S(s_i)$ and $S(t_i)$ in Γ_{μ_i} so that the bounding box of the drawing of each connected component of $G_{\mu_i} - \{s_i, t_i\}$ in Γ_{μ_i}, for $i = 1, \ldots, k$, becomes arbitrarily small with respect to the drawing of $S(s_i)$ and $S(t_i)$. This avoids overlapping between internal vertices of G_{μ_i} and G_{μ_j}, with $i \neq j$, in the next phases of the construction. Then, we scale and translate each drawing Γ_{μ_i} so that $S(t_{i+1}) = S(s_i)$, with $i < k$. It is easy to see that, by the choice of the canonical drawings of each G_{μ_i}, there exists a rectangular region in Γ_μ whose interior does not intersect any square representing a vertex in G_μ^- and whose lower-left corner lies at the intersection point between the vertical line passing through the right side of $S(t)$ and the horizontal line passing through the top side of $S(s)$ in Γ_μ. □

The proof of the next two lemmata also exploits rotations of drawings Γ_{μ_i} and can be carried out in a fashion similar to the proof of Lemma 3.4.

Lemma 3.5. *If μ is an S-node of* `Type B`*, then G_μ admits a pipe drawing.*

Proof. Recall that $\ell(\mu) = 2$, at least one of the children of μ is a bad P-node, say μ_1, and the other child contains an edge between its terminals. The case in which μ_2 is bad and μ_1 is not bad and contains an edge between its terminals can be treated symmetrically.

By Lemma 3.3, we can construct a drawing Γ_{μ_1} of G_{μ_1} and a drawing Γ_{μ_2} of G_{μ_2} such that Γ_{μ_1} is an H1 drawing and Γ_{μ_2} is a V1 drawing, if μ_2 is also bad, or a D1$^\diamond$ drawing, if μ_2 is good or almost bad.

We show how to compose Γ_{μ_1} and Γ_{μ_2} into a pipe drawing of G_μ as follows. Refer to Figure 3.5(d) for an example where Γ_{μ_1} is an H1 drawing and Γ_{μ_2} is a D1$^\diamond$ drawing.

First, we replace Γ_{μ_2} with its copy rotated by $-90°$. Second, we scale $S(s_i)$ and $S(t_i)$ in Γ_{μ_i} so that the bounding box of the drawing of each connected component of $G_{\mu_i} - \{s_i, t_i\}$ in Γ_{μ_i}, with $i \in \{1, 2\}$, becomes arbitrarily small with respect to the drawing of $S(s_i)$ and $S(t_i)$. Third, we scale and translate drawing Γ_{μ_1} and Γ_{μ_2} so that $S(t_2) = S(s_1)$. Finally, we perform an $\overset{\searrow}{s}$-scaling of the obtained drawing of G_μ so that the drawing of G_μ^- lies above the bottom side of $S(s)$. By construction and by the choice of the canonical drawings of G_{μ_1} and G_{μ_2}, the resulting drawing of G_μ is a pipe drawing. This concludes the proof. □

Lemma 3.6. *If μ is an S-node of* `Type C`*, then G_μ admits both a pipe drawing and a rectangular drawing.*

Proof. Recall that $\ell(\mu) = 2$, $s_1t_1 \in E(G_{\mu_1})$, $s_2t_2 \in E(G_{\mu_2})$, and none of μ_1 or μ_2 is bad.

By Lemma 3.3, we can construct a drawing Γ_{μ_1} of G_{μ_1} and a drawing Γ_{μ_2} of G_{μ_2} such that Γ_{μ_1} is an H1$^\diamond$ drawing and Γ_{μ_2} is a D1$^\diamond$ drawing.

We show how to compose Γ_{μ_1} and Γ_{μ_2} into a pipe drawing Γ_P of G_μ; refer to Figure 3.5(e).

The first three steps of the construction are exactly as in the proof of LEMMA 3.5. To obtain Γ_P, we perform an $\overset{\searrow}{s}$-scaling and a $\overset{\swarrow}{t}$-scaling of the obtained drawing of G_μ so that the bottom side of $S(s)$ and the bottom side of $S(t)$ lie below the bottom side of $S(s_1) = S(t_2)$.

Finally, we show how to compose Γ_{μ_1} and Γ_{μ_2} into a rectangular drawing Γ_R of G_μ; refer to Figure 3.5(b). In this case, we obtain Γ_R simply by scaling and translating drawing Γ_{μ_1} and Γ_{μ_2} so that $S(t_2) = S(s_1)$.

It is not hard to see that, by construction and by the choice of the canonical drawings of G_{μ_1} and G_{μ_2}, drawings Γ_P and Γ_R are a pipe drawing and a rectangular drawing of G_μ, respectively. This concludes the proof. □

3.4 Triconnected Simply-Nested Graphs

In this section, we devote our attention to 3-connected simply-nested graphs.

A cycle-tree with a single edge removed from the outer cycle is a *path-tree* (to avoid special cases, we allow the outer cycle of the cycle-tree to be a 2-gon). In path-trees, we refer to vertices in the tree as *tree vertices* and vertices in the external path as *path vertices*. A tree vertex can *see* a path vertex if they share a face in the original cycle-tree.

Define an *almost-triconnected path-tree with root ρ, leftmost path vertex ℓ, and rightmost path vertex r* to be a path-tree containing in one of its faces a tree vertex ρ and path vertices ℓ and r such that if the edges $\rho\ell$, ρr, and ℓr were added, the resulting graph would be a 3-connected cycle-tree.

SPQ-decomposition of path-trees. We now describe a recursive decomposition for almost-triconnected path-trees. We call this an SPQ-decomposition, because it bears a strik-

ing similarity to the SPQ-decomposition of series-parallel graphs. Let G be a 3-connected cycle-tree, let ℓr be an edge incident to the outer cycle of G, and let ρ be a tree vertex incident to the internal face of G edge ℓr is incident to. Also, let $G' = G - \ell r$ be the almost-triconnected path-tree obtained from G by removing edge ℓr. Graph G' defines a rooted decomposition tree T whose nodes are of three different kinds: *S*-, *P*-, and *Q-nodes*. Each node μ of T is associated with a path-tree G_μ with root ρ_μ, leftmost path vertex ℓ_μ, and rightmost path vertex r_μ obtained—except the Q-nodes—from smaller path-trees T_i with root ρ_i, leftmost path vertex ℓ_i, and rightmost path vertex r_i, for $i = 1, \ldots, k$, as follows.

- A *Q-node* μ is associated with a path-tree G_μ with three vertices: one tree vertex ρ_μ and two path vertices ℓ_μ and r_μ. The tree vertex ρ_μ is the root of G_μ, while path vertices ℓ_μ and r_μ are the leftmost and the rightmost path vertex of G_μ, respectively. Edge $\ell_\mu r_\mu$ will always exist, but edges $\rho_\mu \ell_\mu$ and $\rho_\mu r_\mu$ may or may not exist; see Figure 3.6(left).

- An *S-node* μ is associated with a path-tree G_μ obtained from path-tree T_1 by adding a new root ρ_μ connected to ρ_1. Also, $\ell_\mu = \ell_1$ and $r_\mu = r_1$ are the leftmost and the rightmost path vertex of G_μ, respectively. Edges $\rho_\mu \ell_\mu$ and $\rho_\mu r_\mu$ may or may not exist; see Figure 3.6(middle).

- A *P-node* μ is associated with a path-tree G_μ obtained from path-trees T_i by merging $T_1, T_2, \ldots, T_k$ from left to right as follows. First, roots ρ_i are identified into a new root ρ_μ. Then, the rightmost path vertex r_i of T_i and the leftmost path vertex ℓ_{i+1} of T_{i+1}

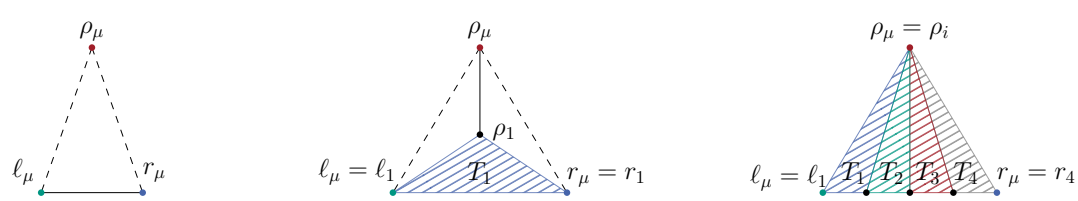

Figure 3.6: Path-trees associated with a Q-node (left), an S-node (middle), and a P-node (right). Dashed edges may or may not exist. Striped triangles represent smaller path-trees T_i with root ρ_i.

are identified, for $i = 1, \ldots, k-1$. Path vertices $\ell_\mu = \ell_1$ and $r_\mu = r_k$ are the leftmost and the rightmost path vertex of G_μ, respectively; see Figure 3.6(right).

We have the following lemma.

Lemma 3.7. *Any almost-triconnected path-tree admits an SPQ-decomposition.*

Proof. We induct on the height of the path-tree G.

If the tree in G has height 1, then the root is the only tree node. If there are only two path vertices, then G is simply a Q-node. Otherwise, it is a P-node. Let the path in G be $p_1, p_2, \ldots, p_k$. Then G can be produced by merging $k-1$ Q-nodes. The child T_i of the P-node will be the Q-node consisting of root r, left path vertex p_i, and right path vertex p_{i+1}.

If the tree in G has height greater than 1, then first suppose that the only path vertices *root* can see are ℓ and r. Then *root* must have exactly one child in T, since otherwise there would be at least one visible path vertex between each pair of adjacent children. So G is an S-node whose single child contains $G - root$. We then continue the decomposition recursively.

Otherwise *root* can see a path vertex $p \neq \ell, r$. Either there is an edge from *root* to p, or there is space to draw such an edge. The vertices *root* and p divide G into two components. Let $p_{i_1}, \ldots, p_{i_k}$ be the left to right sequence of path vertices visible from *root*. For each j, the vertices *root*, p_{i_j}, and $p_{i_{j+1}}$ define a subgraph G_j containing the path vertices inclusively in between p_{i_j} and $p_{i_{j+1}}$ and the tree vertices on paths in the tree from *root* to those path vertices. So G is a P-node with children consisting of G_j for $j = 1, \ldots, k-1$. Each of these children has a smaller height, so we continue the decomposition recursively. □

Lemma 3.8. *Any almost-triconnected path-tree G without separating triangles and whose outer face is not a triangle admits a square-contact representation.*

Proof. As we construct the path-tree using the SPQ-decomposition, we maintain a square-contact representation for each node. For a path-tree H with root vertex $root$, leftmost path vertex ℓ, and rightmost path vertex r, the square-contact representation for H that we construct obeys the following invariant.

Inductive Invariant. There are two main cases we consider: either H has two path vertices or H has more than two path vertices. First if H has only two path vertices, then $root$ may be connected to ℓ or r, but not both. Here we describe the invariant for when $root$ is connected to r. The version where $root$ is adjacent to ℓ is symmetric swapping ℓ with r and left with right throughout. If neither edge is present in H, then we can use either version of the invariant.

1. $S(\ell)$ appears to the left of $S(root)$ and $S(r)$ appears directly to the right of $S(\ell)$ and is below $S(root)$. The bottom of $S(root)$ contacts the top of $S(r)$.

2. The top of $S(\ell)$ is vertically between the bottom of $S(root)$ and one-third the distance to the top of $S(root)$. The bottom of $S(root)$ is vertically above the bottom border of $S(\ell)$.

3. If $root$ and r are adjacent, then all other squares are inside the bounding box defined by extending the lines of the top of $S(\ell)$, the top of $S(r)$, the left of $S(root)$ and the right of $S(\ell)$. We call this the $S(root), S(\ell)$-bounding box.

Figure 3.7 illustrates the two-path-vertex, $root$ adjacent to r, case of the invariant. Note that the square-contact representation can be rotated 90° such that $S(\ell)$ is below $S(root)$ and $S(r)$ appears to the right of $S(root)$ at the top of the square-contact representation. We call the original representation the U orientation and the rotated variant the C orientation for the shape formed by $S(\ell)$, $S(root)$, and $S(r)$. The interior squares are fit into the "concavity" of the letter for the orientation.

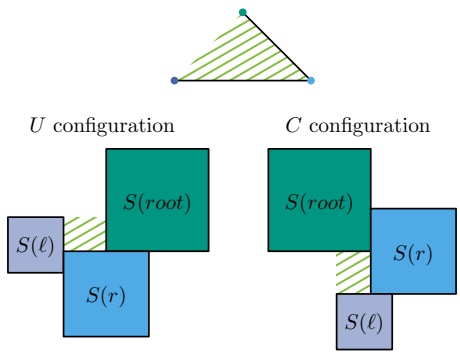

Figure 3.7: The invariant for a cycle-tree with two path vertices.

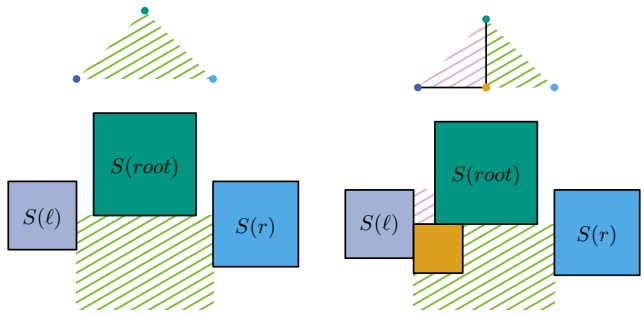

Figure 3.8: The invariant for a cycle-tree with more than two path vertices.

If H has more than two path vertices, then:

1. $S(\ell)$ and $S(r)$ have their top on the same horizontal line.

2. $S(\ell)$, $S(root)$, and $S(r)$ appear in that order from left to right with the tops of $S(\ell)$ and $S(r)$ between the bottom of $S(root)$ and one-third the distance to the top of $S(root)$ and the bottoms of $S(\ell)$ and $S(r)$ are below the bottom of $S(root)$.

3. If ℓ and $root$ are adjacent, then the right border of $S(\ell)$ touches the left border of $S(root)$. If ℓ and $root$ are not connected, then there is a horizontal gap between the right border of $S(\ell)$ and the left border of $S(root)$. This is symmetric for the r and $root$ connection.

4. If H is a P-node and the leftmost (or rightmost) child being merged has only two path vertices, then other squares are allowed inside the $S(root), S(\ell)$-bounding box.

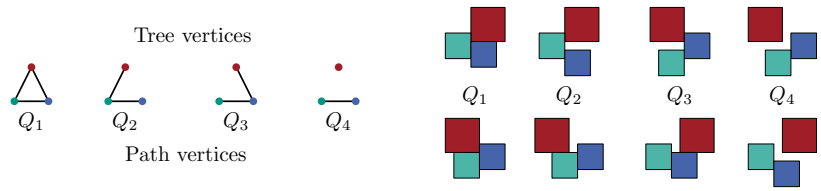

Figure 3.9: The four possible path-trees for a Q-node and their square-contact representations.

5. All other squares are drawn in the region below $S(root)$, to the right $S(\ell)$, and to the left of $S(r)$.

6. Only interior squares contacting $S(root)$ may touch the horizontal line extending out from the bottom of $S(root)$. Only squares contacting $S(\ell)$ may touch the line extending out from the right of $S(\ell)$. Only squares contacting $S(r)$ may touch the line extending out from the left of $S(r)$.

Figure 3.8 illustrates the three-path-vertex case of the invariant. Together the two cases of this invariant guarantees the drawings for S-nodes and P-nodes resembles Figure 3.11.

We now show how to construct a square-contact representation Γ for a path-tree G by inductively constructing a square-contact representation Γ_x that obeys the invariant for each node x in the decomposition. Let $root_x$ be the root vertex of the tree vertices, ℓ_x be the leftmost path vertex, and r_x be the rightmost path vertex for a decomposition node x. We may drop subscripts when the decomposition node is clear.

First, Figure 3.9 shows square-contact representations for each Q-node type obeying the invariant for graphs with two path vertices.

Handling S-nodes If x is an S-node that has just 2 path vertices with child node c, then $root_x$ cannot have edges to both ℓ_x and r_x, otherwise the three vertices form a separating triangle. If $root_x$ is adjacent to ℓ choose the orientation of Γ_c that places $S_{\Gamma_c}(r)$ below $S_{\Gamma_c}(root_c)$. Then construct Γ_x by drawing a square for x of the same size as $S_{\Gamma_c}(root_c)$

directly on top of $S(root_c)$ and perform $\overset{\nwarrow}{\ell}$-scaling until its top is vertically between the bottom of $S(root_x)$ and one-third of the way to the top of $S(root_x)$. Γ_c may not have had an adjacency between $root_c$ and ℓ, in which case we $\overset{\nwarrow}{root}_x$-scale until its left contacts the right of $S(\ell)$. The case when $root_x$ is adjacent to r can be handled symmetrically. Finally when $root_x$ is adjacent to neither ℓ nor r, we can construct a case where there is a $root_x, \ell$ edge and then perform a slight negative $\overset{\nwarrow}{root}_x$-scaling to remove the extra contact.

If x is an S-node with at least 3 path vertices and child node c, then constructing the square-contact representation is much simpler. $\Gamma_c$obeys the second case of the invariant. Therefore in Γ_c, $S_{\Gamma_c}(\ell)$ appears on the left of the drawing and $S_(\Gamma_c)(r)$ appears on the right of the drawing with the other squares appearing between them. To construct Γ_x, start with Γ_c and place a new square for $root_x$ on the top of $S(root_c)$ of equal size. If $root_x$ is adjacent to ℓ, $\overset{\nwarrow}{root}_x$-scale until the left border is vertically in line with the right border of $S(\ell)$. If $root_x$ is adjacent to r, $\overset{\nearrow}{root}_x$-scale until the right border is vertically in line with the left border of $S(r)$. Finally $\overset{\nwarrow}{\ell}$-scale and $\overset{\nearrow}{r}$-scale until their top borders lie between the bottom border of $S(root_x)$ and one-third of the way to the top of $S(root_x)$. It is possible $root_c$ was adjacent to ℓ or r and $root_x$ is not, if so we perform a slight negative $\overset{\nwarrow}{root}_x$ or $\overset{\nearrow}{root}_x$-scaling to remove the contact.

Every contact in Γ_c is preserved in Γ_x, because every square in Γ_c lies below the one-third line of $S_{\Gamma_c}(root_c)$. The only new contacts required in Γ_x are those involving $root_x$ which must contact $root_c$ and may contact ℓ or r. The placement of $S_{\Gamma_x}(root_x)$ guarantees contact with $S_{\Gamma_x}(root_c)$ and the scaling of $S_{\Gamma_x}(root_x)$ introduces contacts with $S_{\Gamma_x}(\ell)$ and $S_{\Gamma_x}(r)$ if needed. By construction Γ_x obeys the invariant.

Handling P-nodes We handle a P-node x by repeatedly merging the square-contact representations for two adjacent children c_1 and c_2. For the pair of graphs being merged, call the shared root vertex $root$, the leftmost path vertex ℓ, the path vertex they share in common

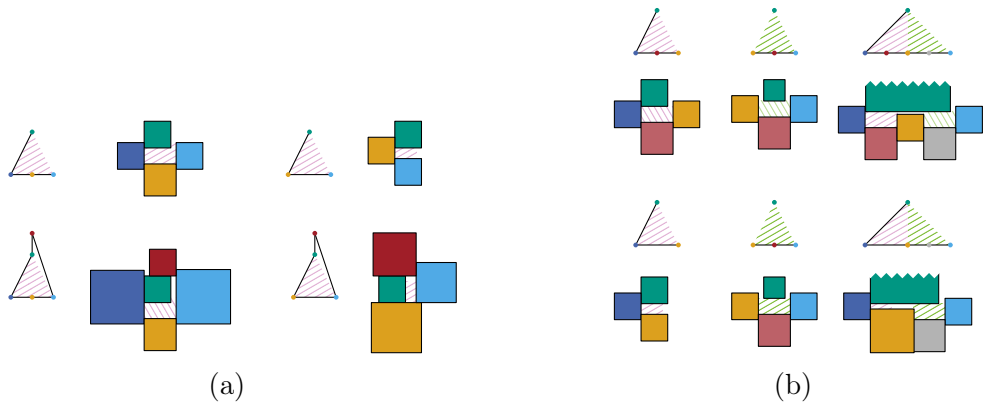

Figure 3.10: Examples of the square-contact construction of an S-node (a) and of a P-node (b).

m, and the rightmost path vertex r. We use the C configuration for every child node with just two path vertices, unless it is the leftmost child and has an edge to the right of the two path vertices or it is the rightmost child and has an edge to the left of the two path vertices.

The order we merge children requires some care. For any child in a C configuration, prioritize merging it in the direction of its "concavity", that is if *root* is adjacent to ℓ then merge the child with its right sibling. We perform all of these prioritized merges before any others.

After recursively constructing Γ_{c_1} and Γ_{c_2}, scale the two representations such that $S_{\Gamma_{c_1}}(m)$ is the same size as $S_{\Gamma_{c_2}}(m)$.

If neither or both of Γ_{c_1} and Γ_{c_2} are C configurations, we perform additional scaling to guarantee the bottom of $S_{\Gamma_{c_1}}(root)$ and $S_{\Gamma_{c_2}}(root)$ are at the same height. To do so, take the side with the higher square bottom for *root* and perform $\overset{\searrow}{m}$-scaling and then rescale down the drawing to keep $S_{\Gamma_{c_1}}(m)$ the same size as $S_{\Gamma_{c_2}}(m)$. Now replace the squares for *root* with a new square sharing bottom left corners with $S_{\Gamma_{c_1}}(root)$ and bottom right corners with $S_{\Gamma_{c_2}}(root)$. In some cases, this new square overlaps the top interior of $S(m)$. So if it is needed, we translate $S(m)$ downwards either until its top is in line with the bottom of $S(root)$, if m and *root* are adjacent, or until it is slightly below $S(m)$, if m and *root* are not

adjacent. The resulting square-contact representation has at least three path vertices and obeys the second case of the invariant.

Because of our merge ordering, if there is only one C configuration then *root* must not be adjacent to m. If we attempted to perform the above procedure with such a C configuration, then the result would not obey our invariant because the square of an outer path vertices would be below $S(root)$.

When only one of Γ_{c_1} and Γ_{c_2} is a C configuration and *root* is not adjacent to m, after overlaying $S_{\Gamma_{c_1}}(m)$ and $S_{\Gamma_{c_2}}(m)$ it is not easy to replace the *root* squares with one larger square. Without loss of generality we assume Γ_{c_1} is the C configuration. In this case, we observe that because *root* and m are not adjacent, $S_{\Gamma_{c_2}}(m)$ can be translated downwards to slightly below $S_{\Gamma_{c_2}}(root)$ without disturbing any square contacts. After performing this translation, we can $\overset{\searrow}{m}$-scale followed by rescaling down Γ_{c_1} to keep $S_{\Gamma_{c_1}}(m)$ and $S_{\Gamma_{c_2}}(m)$ the same size such that the bounding box containing the squares of interior vertices has height equal to the slight vertical distance between $S(m)$ and $S(root)$ in the other drawing. Now we can finish the construction by creating a new large square for *root* in the same manner as the previous case.

The line $S(m)$ is scaled along in the P node construction is guaranteed by the invariant to only intersect squares that are already in contact with $S(m)$. Similarly the bottom of the new $S(root)$ sits along the line touched only by the tops of squares of interior vertices adjacent to *root*. The translation of $S(m)$ is also designed such that any square previously in contact with it, will still be in contact after the translation. Therefore Γ_x is a square-contact representation for x. By construction Γ_x follows the invariant.

Figure 3.10 shows our construction for two S-nodes on the left and two P-nodes each with two children on the right. □

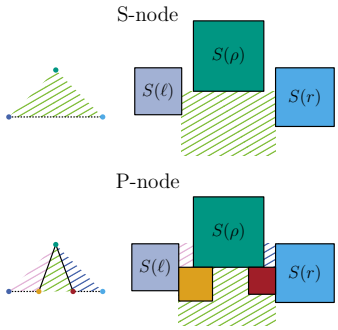

Figure 3.11: Invariants for S- and P-nodes with more than two path vertices.

To construct a square-contact representation for a 3-connected cycle-tree, it is natural to remove an edge in the outer cycle to obtain a path-tree, use Lemma 3.8 to construct a square-contact representation, and then attempt to reintroduce a contact for the removed edge. However, because Lemma 3.8 places the leftmost and rightmost path vertices on the left and right side of the drawing, it is unclear how to add a contact between them. Instead, we split the cycle-tree into two overlapping almost-triconnected path-trees, obtain their square-contact representations by Lemma 3.7, and overlay them to form a square-contact representation for the entire cycle-tree.

Lemma 3.9. *Any 3-connected cycle-tree G without separating triangles and whose outer face is not a triangle admits a square-contact representation.*

Proof. Given a 3-connected cycle-tree graph G without separating triangles, we split it into two path-trees using the following method:

1. Root the tree in G arbitrarily.
2. Let v be a leaf vertex of the tree in G.
3. If v can see at least three path vertices, select v and a subsequence of three path vertices visible from v.
4. Otherwise travel upwards in the tree until reaching a vertex u that can see at least three path vertices

5. There are at most two path vertices connected to descendants of u in the tree. Select u and a subsequence of three path vertices visible from u that include the path vertices connected to the descendants of u.

6. Now in either case, we have selected one tree vertex r and three consecutive path vertices which in counter-clockwise order are p_1, p_2, and p_3 visible from r. Removing r, p_1, and p_3 disconnects the cycle-tree into two subgraphs H_1 and H_2 where H_1 is the subgraph containing p_2.

7. $G - H_1$ and $G - H_2$ are two path-trees that coincide on r, p_1, and p_3.

Figure 3.7 depicts using this method to obtain two path-trees from a cycle-tree. Because the outer face of G has at least four path vertices, $G - H_2$ has exactly three path vertices, and the two subgraphs share two path vertices, $G - H_1$ has at least three path vertices. The two graphs $G - H_1$ and $G - H_2$ are almost-triconnected path-trees and so by Lemma 3.7, both graphs can be constructed by our decomposition. Then using r as the root of both path-trees, Lemma 3.8 guarantees $G - H_1$ and $G - H_2$ have square-contact representations Γ_1 and Γ_2 respectively satisfying the second case of the invariant.

In particular, Γ_1 has $S_{\Gamma_1}(r)$ in between $S_{\Gamma_1}(p_3)$ on the left and $S_{\Gamma_1}(p_1)$ on the right while Γ_2 has $S_{\Gamma_2}(r)$ in between $S_{\Gamma_2}(p_1)$ on the left and $S_{\Gamma_2}(p_3)$ on the right. Our goal is to align the squares for r, p_1, and p_3 in both drawings while not introducing in new square-contacts.

We construct such a square-contact representation $\Gamma = \Gamma(G)$ as follows:

1. Let $d_i(x, y)$ be the horizontal distance between $S_i(x)$ and $S_i(y)$.

2. Scale the entire drawings of Γ_1 and Γ_2 so that $d_1(p_1, p_3) = d_2(p_1, p_3)$.

3. Rotate Γ_2 by 180° and vertically align the right sides of the squares $S_{\Gamma_1}(p_1)$ and $S_{\Gamma_2}(p_1)$. Note that the left sides of the squares $S_{\Gamma_1}(p_3)$ and $S_{\Gamma_2}(p_3)$ are also vertically aligned.

4. If $d_1(r, p_1) = d_2(r, p_1)$ and $d_1(r, p_3) = d_2(r, p_3)$, then $S_{\Gamma_1}(r)$ and $S_{\Gamma_2}(r)$ are the same size and we translate Γ_2 vertically such that $S_{\Gamma_1}(r)$ and $S_{\Gamma_2}(r)$ overlap exactly.

5. If $d_1(r, p_1) \neq d_2(r, p_1)$ and without loss of generality $d_1(r, p_1) < d_2(r, p_1)$, then perform $\overset{\swarrow}{r}$-scaling in Γ_2 until the distances are equal. If there were squares in the $S_{\Gamma_2}(r), S_{\Gamma_2}(p_1)$-bounding box, then perform $\overset{\searrow}{a}$-scaling on the squares in the bounding box where a is the bottom right corner of the bounding box until the horizontal width of the bounding box is $d_1(r, p_1)$.

6. If $d_1(r, p_3) \neq d_2(r, p_3)$, perform the analogous scaling to make them equal too.

7. Perform $\overset{\searrow}{p_1}$-scaling so that its top border is above the top of $S_{\Gamma_2}(p_1)$. Do the analogous scaling for $S_{\Gamma_1}(p_3)$.

8. Remove $S_{\Gamma_2}(r)$, $S_{\Gamma_2}(p_1)$, and $S_{\Gamma_2}(p_3)$.

After the final removal step, Γ has one square for each vertex in G. We prove for each edge uv in $G - H_1$ that $S_\Gamma(u)$ contacts $S_\Gamma(v)$ using some case analysis, but first observe that rotations and scalings of entire square-contact representations preserve square contacts so we only need consider cases where u or v was scaled. The only squares undergo scaling in Γ_1 or Γ_2 are the squares for r, p_1, p_3, and any vertices in the $S_i(r), S_i(p_j)$-bounding boxes.

- Without loss of generality if u is p_1 (or p_3) then $S_{\Gamma_1}(v)$ touches the right side of $S_{\Gamma_1}(p_1)$. When $S(p_1)$ is scaled, the resulting right border is an extension of the original. The left border of $S(v)$ contains $S_{\Gamma_1}(v)$, because any scaling $S_{\Gamma_1}(v)$ underwent (possibly because $v = r$ or v is in the $S_{\Gamma_1}(r), S_{\Gamma_1}(p_1)$-bounding box) holds at least one point in contact with $S_{\Gamma_1}(p_1)$.

- If u is r and v is not p_1 or p_3, then v may be below r or in the $S_{\Gamma_1}(r), S_{\Gamma_1}(p_1)$-bounding box. If v is below r, then the contact is also preserved, because whenever $S(r)$ is scaled the bottom is a superset of the previous bottom.

This argument is nearly symmetric for an edge in $G - H_2$. The only difference to observe is that the squares $S_{\Gamma_2}(p_1)$ or $S_{\Gamma_2}(p_3)$ are removed in favor of $S_{\Gamma_1}(p_1)$ and $S_{\Gamma_1}(p_3)$. Other squares only contact $S_{\Gamma_2}(p_1)$ on the right and $S_{\Gamma_2}(p_3)$ on the left. After scaling $S_{\Gamma_1}(p_1)$ ($S_{\Gamma_1}(p_3)$), the right (left) border $S_{\Gamma_1}(p_1)$ ($S_{\Gamma_1}(p_3)$) contains the right (left) border of $S_{\Gamma_2}(p_1)$ ($S_{\Gamma_2}(p_3)$). Therefore every edge in G has proper contact in Γ.

We also observe that no new contacts were introduced by these steps. Because for $i = 1, 2$ and $j = 1, 3$ the top border of the $S_i(t), S_i(p_j)$-bounding boxes are below the one-third line on $S_i(t)$, the overlaying of $S_{\Gamma_1}(t)$ with $S_{\Gamma_2}(t)$ does not introduce any new contacts (other than overlapping squares for the same vertex).

Thus Γ is a proper square-contact representation of G. □

As Halin graphs are 3-connected cycle-trees without separating triangles and have, except for K_4, a non-triangular outer face, we have the following.

Corollary 3.1. *Any Halin graph $G \not\simeq K_4$ admits a square-contact representation.*

Next, we investigate square-contact representations of 2-outerplanar simply-nested graphs that are not cycle-trees (Theorem 3.3) and 3-outerplanar simply nested graphs (Theorem 3.4).

Theorem 3.3. *There exists a 3-connected 2-outerplanar simply-nested graph that does not admit any proper square-contact representation.*

Proof. Consider the two nested quadrilaterals shown in Figure 3.12(left). One of its two quadrilateral faces must be the outer one, giving the embedding shown. In any square-contact representation, the inner polygon surrounded by the squares for the four outer vertices must be a rectangle, as it has only four sides. Each of the four inner squares must

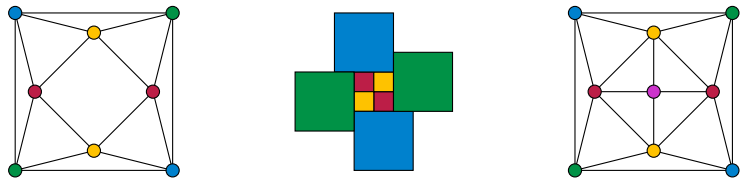

Figure 3.12: Left: Two nested quadrilaterals form a graph with no proper square-contact representation. Middle: An improper square-contact representation for the same graph. Right: A graph with no square-contact representation, even an improper one.

touch one of the four corners of this rectangle (the corner made by its two outer neighbors). For the four inner squares to touch the four corners of the rectangle and each other, the only possibility is that the rectangle is a square and each inner square fills one quarter of it, as shown in Figure 3.12(middle). However, this representation is improper, as diagonally-opposite inner squares meet at their corners. □

Theorem 3.4. *There exists a* 3*-connected* 3*-outerplanar simply-nested graph that does not admit any square-contact representation.*

Proof. Consider the graph shown in Figure 3.12(right). Its quadrilateral face must be the outer one, giving the embedding shown. As in the proof of Theorem 3.3, the only possible representation for its two outer quadrilaterals has the four outer squares surrounding a central square region, divided into four quarters representing the four middle vertices, as shown in Figure 3.12(middle). However, this representation leaves no room for the inner vertex. □

We remark that the graph of Theorem 3.4 is actually 2-outerplanar simply-nested, but not with its quadrilateral face as the outer face.

3.5 Conclusions

In this chapter, we provided simple characterizations for two notable families of planar graphs that admit proper square-contact representations. Moreover, we introduced a new decompo-

sition for an interesting family of polyhedral graphs that generalize the Halin graphs, i.e., the 3-connected cycle-trees. Finally, we showed that the absence of separating triangles and a non-triangular outer face do not guarantee the existence of weak and proper square-contact representations of 3-outerplanar and 2-outerplanar simply-nested graphs, respectively.

Chapter 4

J-Viz: Finding Algorithmic Complexity Attacks via Graph Visualization of Java Bytecode *

4.1 Introduction

The Space/Time Analysis for Cybersecurity (STAC) program [26] at the U.S. Defense Advanced Research Projects Agency (DARPA) aims to develop new program analysis techniques and tools for identifying vulnerabilities related to the space and time resource usage behavior of algorithms, and specifically to vulnerabilities based on algorithmic complexity and side channel attacks. STAC seeks to enable security analysts to identify algorithmic resource usage vulnerabilities in software to support a methodical search for them in the software upon which the U.S. government, military, and economy depend [26].

*Portions of this chapter are included from [2].

4.1.1 Our Contributions

In this chapter, we describe a tool, the JVM abstracting abstract machine (Jaam) Visualizer, or "J-Viz" for short, which is intended for use by security analysts to perform such searches through the exploration of graphs derived from Java bytecode. Thus, we are not attempting to solve the problem of identifying software algorithmic complexity attacks in a completely automated way, but instead we are providing a means for doing semi-automated analysis that increases the efficiency of a human analyst. The workflow for our tool involves taking a given program, specified in Java bytecode, and constructing a control-flow graph of the possible execution paths for this software, using a framework known as *control flow analysis* (CFA) [57]. Our tool then provides a human analyst with an interactive view of this graph, including heuristics for aiding the identification of which parts of the provided program seem suspicious.

One of the main components of our J-Viz tool involves visualizing control-flow graphs in a canonical way based on a novel vertex numbering scheme that we call the *sibling-first recursive* numbering. This numbering scheme is essentially a hybrid between the well-known breadth-first and depth-first numbering schemes, but differs from both in a way that appears to be more useful for visualizing control-flow graphs so as to highlight potential algorithmic complexity attacks. In particular, as we show that with respect to some actual test cases provided to us by DARPA that this approach is effective at aiding a human security expert to find such attacks. We designed J-Viz with the following goals in mind:

- We want users to be easily able to recognize patterns in source code from our visualizations. Thus, similar sections of code should produce similar subgraphs, which should be drawn in a similar way.
- We want to use a hierarchical visualization, in which users can collapse or expand sections of the graph to different levels of detail. But we also want them to be able to

build a consistent mental model of the graph. Thus, drawings should not drastically shift the relative positions of the vertices when sections are collapsed or expanded.

- No matter what sequence of actions the user performs, drawings should be consistent. That is, the same view of a graph, in which the same set of nodes are collapsed and expanded, should always be drawn in the same way.

- Our system should rank sections of the graph by how likely they are to produce vulnerabilities, and display this information visually to the user.

4.1.2 Related Work

Although it is using different means to achieve mental map preservation, the J-Viz system follows in a long line of research on techniques directed at preserving the mental map of a graph drawn dynamically. For instance, Misue *et al.* [49] discuss node movement adjustments, including avoiding node overlaps, for preserving the mental map. Diehl and Carsten [19] discuss force-directed approaches for preserving the mental map between instances of a changing graph. Goodrich and Pszona [34] study efficient algorithms for minimizing vertex movements as a graph is incrementally revealed in an online manner. With respect to existing software systems, the Graphviz [21] and GraphAEL [23] systems both include algorithms intended to preserve the mental map as a graph is modified. Bridgeman and Tamassia [10] formally study metrics for characterizing mental map preservation between different instances of a changing graph. So as to provide an empirical basis for such work, a user study of Purchase *et al.* [53] supports the thesis that preserving the mental map for graph visualization is a useful goal to aid users in performing tasks on graphs.

Visualization tools have also previously been applied to source code. Doxygen [61], a tool for automatically generating documentation, can produce various kinds of diagrams for visualizing code, including call graphs. It is generally configured to use the *dot* [30] tool from

GraphViz to draw these graphs hierarchically. Similarly, Visual Studio can visualize call graphs to aid programmers in debugging applications [17]. In constrast with these systems, our J-Viz tool provides four main features that these tools do not provide. First, J-Viz shows a greater level of detail, since it analyzes code at the level of individual instructions rather than methods. Second, J-Viz allows the user to interact with a graph and produce multiple views of the same Java bytecode. Third, the layout algorithm used in J-Viz is designed to draw similar code fragments in the same (canonical) way, so as to highlight portions of repeated code. Fourth, J-Viz guides the user to potential security vulnerabilities, by highlighting nodes that are believed to be risky based on algorithmic complexity (or other factors), whereas these other systems were not focused on software security.

Another tool, Jinsight [16], can be used to profile a Java program to provide various views of resource usage, such as highlighting which instances of a class have taken the most time or used the most memory. This tool does not provide a full graph of the program's possible execution paths, however, which we believe to be essential for detecting security vulnerabilities. In addition to these Java-based tools, there are also tools for visualizing compiled executibles in other languages for malware analysis, such as the tool by Quist and Liebrock [54].

At a high level, our work is also related to recent work on applying graph visualization tools for security visualization. For example, work by Di Battista *et al.* [7] on visualizing flows in the Bitcoin transaction graph and work by Mansman *et al.* [45] on visualizing host behavior in a network are also in this area. (See also the surveys by Tamassia *et al.* [60] and Wagner *et al.* [62].)

4.2 Graph Generation via Static Analysis

In this section, we review the process that takes Java bytecode as input and produces the graphs that are visualized in J-Viz. These graphs are produced using the *JVM abstracting abstract machine* (*Jaam*) tool [13] developed at the University of Utah based on the work of Van Horn and Might [43], which itself is based on control-flow analysis (CFA) framework known as *k-CFA* [46, 57]. Because there could be exponentially many possible execution paths of any given program, which would be too large to visualize and reason about, the k-CFA framework compresses execution paths into a graph of reasonable size that represents possible executions of a Java program at the instruction level. Such a graph is called *sound* if it represents every possible execution path, and *precise* if it excludes every impossible execution path. The k-CFA framework is sound, and it has a tunable degree of precision based on the integer parameter, k, albeit at the cost of creating additional states in the graph for larger values of k.

At the lowest level of the hierarchy, 0-CFA, we discard contextual information and generate one state, which forms a vertex in the graph representation, for each line of Java bytecode. Then we add edges for every possible state that could be reached from a given state. For example, a return statement will have an edge to every place from which our current method could have been called. At the next level, 1-CFA, each state also tracks the location from which its method was called. (See Fig. 4.13 in Section 4.6.2 for example graphs produced by 1-CFA.)

This easily generalizes to higher levels, so that for k-CFA, each state stores the locations of the previous k function calls. This added information allows many of the spurious branches produced by 0-CFA to be pruned. (For additional information, please see more detailed descriptions of k-CFA [43, 46, 57].)

0-CFA is known to take $O(n^3)$ time to construct a graph for a program with n lines of code, and this is believed to be tight [38]. k-CFA is EXPTIME-complete for functional languages [42], but can be solved in polynomial time for object-oriented languages [47]. Thus, to provide a reasonable balance between soundness, precision, and efficiency, the version of the Jaam static analyzer used for the work of this chapter is based on 1-CFA. To summarize, then, the Jaam static analyzer takes as input Java bytecode for a given program and produces an directed graph, G, that represents the results of a 1-CFA performed on this bytecode. This graph is *ordered*, in the sense that the outgoing edges for each node are sorted according to the order in which the corresponding instructions appear in the Java bytecode.

4.3 Our Sibling-First Recursive Layout Algorithm

Our approach to the layout of graphs produced by the Jaam tool [13] is based on what we believe is a novel graph numbering scheme, which we call a *sibling-first recursive* (SFR) numbering. Intuitively, SFR is a hybrid numbering scheme that combines features of a breadth-first search (BFS) numbering and a depth-first search (DFS) numbering. We show in Figure 4.1 the difference between algorithms for doing a depth-first search (DFS) ordering of a directed graph and a sibling-first recursive (SFR) numbering. See also Figures 4.8, 4.9, and 4.12, which illustrate SFR spanning trees and their differences with DFS and BFS spanning trees.

Our motivation for using the SFR numbering is that we feel it produces a rooted spanning tree that corresponds more intuitively with the way that programmers conceptualize the main "backbone" of the control flow of their software. For example, it places the true-false branches of if statements as children of the condition that branches to them. In addition, it places the multiple branches of a switch statement as children of the condition that branches to them,

```
Algorithm DFS(v):
Number v as vertex n

Increment n
for each directed edge
(v, w) do
  if w is not num-
  bered then
    Identify w as a
    child of v
    DFS(w)
```

```
Algorithm SFR(v):
for each directed edge
(v, w) do
  if w is not num-
  bered then
    Identify w as a
    child of v
    Number w as
    vertex n
    Increment n
for each directed edge
(v, w) do
  if w is a child of v
  then
    SFR(w)
```

Figure 4.1: DFS and SFR algorithms to explore the connected component of a vertex, v, in a directed graph, G. We assume there is a global variable, n, which is used to number the vertices. In the case of DFS, we initialize $n = 1$ and call DFS(v) on a vertex v that is to become the root of the DFS tree. In the case of SFR, we initialize a vertex, v, as the root of the SFR tree, numbering it as vertex 1, and we set $n = 2$ and call SFR(v).

even if some of the branches flow-through to other branches. (E.g., see Fig. 4.2.) Furthermore SFR numbering also enables the viewer to visually identify isomorphic subgraphs of the graph, corresponding to identical, repeated or equivalent lines of code; see Fig. 4.10 in Section 4.6.1.

4.3.1 High-Level Description of Our Layout Algorithm

At a high level, there are five steps in our algorithm for producing a drawing of the graph, G:

1. We construct an SFR numbering and rooted spanning tree, T, for our input graph, G, which will be used as the "backbone" of our drawing.

2. We draw the tree T using a recursive placement algorithm.

3. We add the edges of G that are not in the tree T.

4. We highlight in our drawing the sections of our graph that are most likely to contain vulnerabilities, based on various criteria.

5. We automatically group subsets of nodes before displaying the entire graph to the user, in a way that allows the user to expand such collapsed nodes.

In the remainder of this section, we describe in more detail each of the above steps in our layout algorithm.

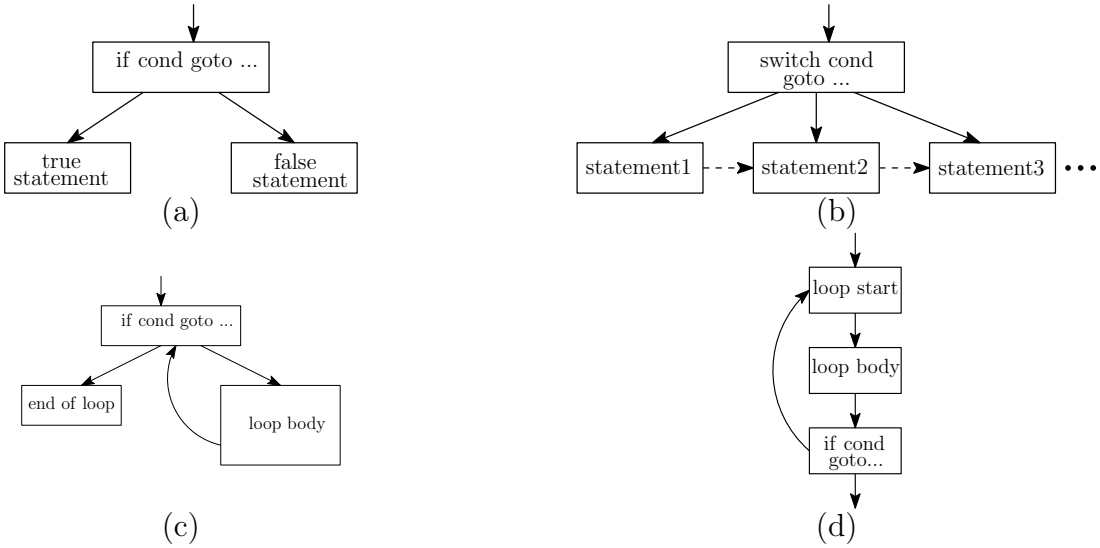

Figure 4.2: Sample graph layout for (a) if-else conditional statement, (b) switch statement (dashed edges for flow through statements), (c) for and while loop, (d) do-while loop.

4.3.2 Constructing an SFR search tree

In our first step, we construct a rooted (ordered) SFR spanning tree, T, for the graph, G, produced from the Jaam tool [13]. The algorithm we use to perform this construction is exactly the SFR algorithm shown in Fig. 4.1, with the added detail that as we traverse the graph G to construct our SFR search tree, T, we process the out-edges from each vertex using the ordering for G, consistent with the intuitive way programmers naturally organize

branches for different types of software branch points. As we highlight in one of our test cases, this approach tends to produce almost identical drawings for repeated (e.g., cut-and-pasted) software code fragments. It also produces ordered combinatorial layouts for each of the following types of code constructs, as shown in Fig. 4.2.

- If-else conditional statement: the true and false components are siblings, with the true component coming first.
- Switch conditional statement: the different branches of the switch statement are siblings of the conditional statement, ordered by their appearance in the code (even if there are non-tree edges between them that would be representing flow-throughs from one branch to another).
- While/for loop: the end-of-loop statement and loop body are both siblings of the conditional statement, with the end-of-loop statement coming first.
- Do-while loop: the start statement, loop body, and conditional statements are in a single path, with a non-tree edge leading back to the start statement.

4.3.3 Drawing the Nodes of our SFR Search Tree

Once we have constructed our (ordered) SFR search tree, T, we draw it recursively, starting from the leaves of T. Each parent is drawn on a row above all of its children. Chains of nodes are drawn in a vertical column. When we reach a branch point, we lay out each of the subtrees from left to right. We require that only a direct descendant of a node can be placed directly underneath it. This means that each subtree, no matter its size, will have an entire vertical lane reserved for it from top to bottom in our graph. See Fig. 4.3.

This requirement might at first seem to waste space in our drawing, but it maintains consistency when a user expands or collapses connected sections of nodes. To see why this is

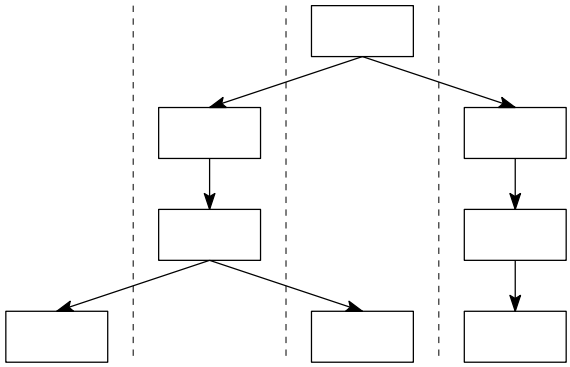

Figure 4.3: A tree drawn according to our algorithm. The dashed vertical lines show separation of subtrees into unique lanes.

so, suppose that two nodes are placed at the same x-coordinate, but neither is an ancestor of the other. Suppose further that the user then chooses to collapse the set consisting of the path from each of these nodes up to their lowest common ancestor. If this happens, then if we simply shift up that portion of the spanning tree, then we will cause overlaps, which would require shift of nodes to fix. (See Fig. 4.4.) But such a shift would be detrimental to the mental map. Thus, rather than produce a compact drawing that reuses vertical space, we use the scheme described above, which tends to preserve the mental map even as we would be collapsing or expanding paths in the spanning tree, T, and shifting the remaining portions accordingly.

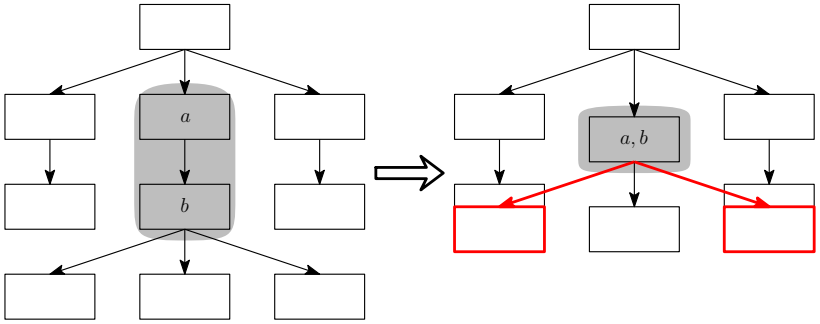

Figure 4.4: An example of how breaking our drawing style can cause problems. When the user collapses nodes a and b, some of the nodes need to be shifted. To avoid this problem, we forbid a node from lying directly above a node that is not a direct descendant.

4.3.4 Drawing Edges

After we have placed the nodes of our ordered spanning tree, T, we must draw all of the edges in our graph. In our case we choose to draw downward edges as straight line segments, and we then draw upward edges as curved segments. In addition to drawing arrows at the ends of such segments, this convention provides a visual cue for which direction an edge is pointing. It also prevents upward edges from lying on top of downward edges. For example, this makes the drawing of the graph of software implementing the bubblesort algorithm, shown in Fig. 4.11 in Section 4.6.2, more readable.

4.3.5 Highlighting

After the nodes and edges of the graph, G, are placed, we highlight vertices to guide the user on where to begin examining it to discover possible security vulnerabilities. For attacks based on increasing the running time for the software on certain inputs, we want to highlight nodes that are likely to be visited the greatest number of times during an execution. So we color our states from green to red based on how likely each node is to be involved in a vulnerability. We have considered the following three ranking methods:

- We could perform a recurrence analysis that computes an upper bound on how many times each node is visited. Automated recurrence analysis has progressed to the point of being able to provide strong upper bounds on many simple algorithms [3]. But we do not believe that this is yet possible for programs such as the ones we need to examine, which contain thousands or tens of thousands of lines of code, and have a complex loop structure.

- We could profile our code, by providing a sample input and counting how many times each state is visited. But while this may help for honestly written programs, we believe

it is unlikely to identify the kinds of deliberate attacks that we need to discover. The programs we are given should in most cases perform well, because otherwise they would not be used at all. But some may have a hidden trigger that causes them to run for much longer.

- We could count the number of nested loops that contain each node. This is a somewhat naive method, but it does seem to give a reasonable heuristic, as will be seen in our experimental results. It is also feasible to compute even for very large graphs. Hence this is the method that we choose to use.

In order to do this highlighting, of course, we need to determine for each node its level of nesting with respect to the loops of the program. We use an adaptation to the SFR tree of a definition by Havlak [37] for DFS trees:

- The outermost loops are the maximal strongly connected components of the directed control-flow graph, G.
- The header for a loop is the first node in the loop that is reached in the SFR tree, T.
- The inner loops are the maximal strongly connected components that remain when the header is removed.

A graph is said to be *reducible* if every cycle has a single entry point [37]. If the graph is reducible, then the loop decomposition does not depend on which rooted spanning is used. But if a cycle has multiple entry points, then the order in which we explore the branches could matter. Thus, in our case, we use the canonical SFR tree, T, that has already been defined for our graph.

To compute loop headers efficiently, we use an algorithm from Tao *et al.* [63]. This traverses the ordered spanning tree and passes loop header information up the tree. While their

method could take a long time for artificially complex graphs, it takes linear time for most real-world programs, because the spanning trees for such graphs tend to be reducible or "nearly" reducible.

4.3.6 Grouping Nodes

We have included four different ways in which nodes in the graph, G can be aggregated, and we present the initial drawing of G to the user based on a pre-defined grouping of certain nodes, with some of these pre-collapsed. First, we choose not to explore nodes that correspond to calls to the Java library, since we do not expect it to contain vulnerabilities. Instead, every such call is collapsed to a single line. This prevents us from creating hundreds of thousands of nodes for the Java library. Nevertheless, the static analyzer, Jaam, needs to do this carefully, so that it can approximate the state of Java library objects and predict their later behavior. For example, any object that is added to an ArrayList or a HashMap can later be taken out. Still, we assume that such an identification is given as an annotation to the input graph, G, since this identification is solely the domain of the Jaam tool. Second, we automatically group each connected set of nodes that belong to the same method. That way, if the user is not interested in the details of a given method, they can collapse it to a single node. Third, we automatically group chains of method nodes that were created in the previous step. Generally, having long chains of nodes taking up a large portion of the screen space hinders the user from seeing the branching structure that they need to find. Finally, we allow the user to select a connected set of nodes and collapse them dynamically, along with providing a comment explaining the purpose of the corresponding section of code.

4.4 The User Interface

In constructing the user interface for the J-Viz system, we relied in part on a survey study of Basil and Keller [6] for software visualizations. They found that the following criteria were considered "absolutely essential" by a majority of participants:

- Search tools for graphical and/or textual elements
- Source code visualization
- Hierarchical representations
- Use of colors
- Source code browsing
- Navigation across hierarchies
- Easy access from the symbol list of the corresponding source code.

To make our system more user-friendly, we have implemented each of these features. We have already discussed our hierarchy for collapsing and expanding vertices, and our use of colors for highlighting nested loops in our graph. We show the other features in use in the screenshot shown in Fig. 4.14 in Section 4.6.3, and describe each of them below.

4.4.1 Searching

We have included the following searching capabilities in our graph:

- Find nodes by SFR numbering
- Find incoming and outgoing edges for a given node
- Find nodes whose methods contain a given string
- Find nodes whose instructions contain a given string

4.4.2 Context panel

When nodes are selected by the user, the code for their methods are shown in the left panel, with the lines for each node highlighted. Alternatively, a user can select lines from the left panel, and we highlight the nodes in our graph that correspond to those lines of code. The lines of code for the innermost loops inside each method are also highlighted.

4.4.3 Description panel

The right panel displays detailed descriptions for each of the selected nodes in our graph.

4.4.4 Minimap

A minimap is given in the lower left that always shows our full graph. When we zoom in on part of the graph, it is highlighted on the map, so that the user never loses track of where they are.

4.5 Experiments

In this section, we describe some test cases we performed to test the effectiveness of the J-Viz system for visualizing Java bytecode and identifying security vulnerabilities that could be triggered by algorithmic complexity attacks. As input to these test cases, we were provided by DARPA with programs to analyze to test our system, some of which were produced by a "red team" tasked with deliberately creating software that contains vulnerabilities to algorithmic complexity attacks.

Our first test case is for a program for verifying a secret password, without revealing any information about such a password. In this case, J-Viz was effective in leading a security analyst to a nested loop that checks each character in the password one at a time. In part, because of the way that the SFR spanning tree lays out conditional branches in an intuitive manner and draws loop edges as curved segments, the analyst was able to notice that the password-checking program exits as soon as it finds the first character that does not match. (See Fig. 4.15 in Section 4.6.3.) The analyst then correctly identified this as a vulnerability (inserted by the red team), since, by timing multiple executions of the program, an attacker can easily determine how many correct characters of a password that they entered with each attempt. Thus, such an attacker could quickly crack the password by a simple iterative search.

Our second test case is for a program for analyzing and classifying images based on features, such as the number of edges or the amount of each color that is present. This program is around 1,000 lines of Java code, and produces about 3,000 nodes in our graph. The goal of the security analyst in this case was to determine if this program can be made to take much longer than it should (specifically, greater than 18 minutes to analyze a 70 KB image). For most images of that size, the program takes around 6 minutes. But, through the use of J-Viz, a security analyst was able to create an image that would take over an hour to be analyzed. The key insight for the analyst was to pay attention to the highlighting in our visualizer, which showed the deepest nested loops in dark red. (See Fig. 4.5.)

In this case, the analyst noticed that many of these nodes were part of the Mathematics class, which provided custom implementations of various mathematical functions. In particular, the exponential function was implemented using a Taylor series, with the number of terms depending on a function of the RGB value of each pixel. This function contained a "spike" which is shown in Figure 4.6, implying a large number of terms in the Taylor series for an image having a particular RGB value. Given this information, the analyst was then able

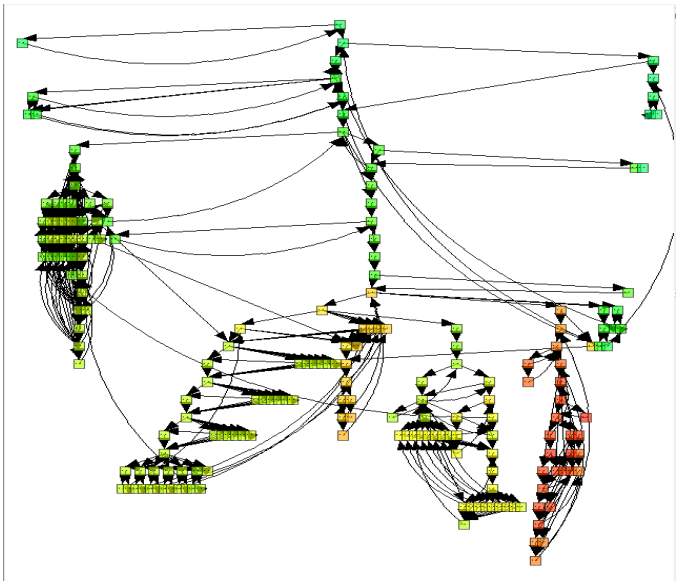

Figure 4.5: Our visualization for the graph for an image processor program, showing the red section of a deeply nested loop (which contained a security vulnerability) in the bottom right.

to create an image that triggered this behavior for every pixel, which took over an hour to process, confirming the vulnerability.

Our final test case is a text analysis program, called *TextCrunchr*, that analyzes documents to compute statistics, such as word frequencies and word lengths, and can process plain text files as well as compressed files (in the case of compressed files, it uncompresses them before analyzing their contents). For example, researchers at Zooomblr are using it in the back-end to analyze a large corpus of data from the Internet.

This program produced about 5,000 nodes in its CFA graph. The goal of the security analyst using our tool was to determine if this program can be made to run for a longer period of time than expected (in this case, greater than 5 minutes to analyze at most 400 KB of data), which could be an indication of a vulnerability to an algorithmic complexity attack.

In using our tool for this task, the key insight for the analyst was to pay attention to the highlighted nodes in our visualizer, which showed the most deeply nested loops in dark red.

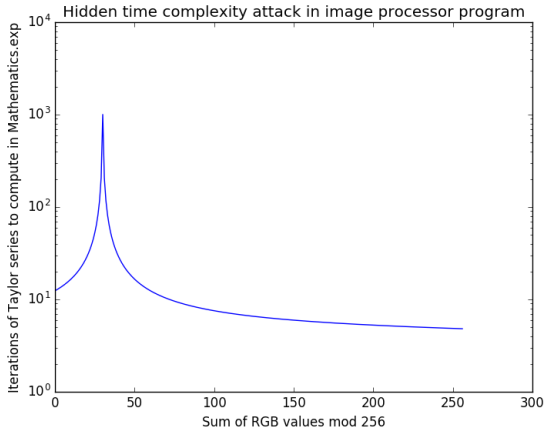

Figure 4.6: The time complexity of image processor spikes by two orders of magnitude when the sum of the RGB values is exactly 30.

In this particular case, the analyst noticed that many of these nodes were part of a custom implementation of a HashMap class that uses some simple bitwise operation to hash an input value. This implementation of Hashmap performed poorly when many hashcode collisions occur, and any input containing many words that hash to the same hashcode triggered quadratic behavior in the text analysis. Given this insight, the analyst created an input file within the 400 KB that forced the program to run significantly longer than 5 minutes, thus uncovering the algorithmic complexity attack vulnerability.

In addition to these test cases, we show additional examples and screenshots of our tool in Sections 4.6.2 and 4.6.3.

4.6 Additional illustrations

4.6.1 SFR Numbering

Fig. 4.8 illustrates a graph (for the complete code of a bubblesort implementation); the vertices are labeled with their SFR numbers.

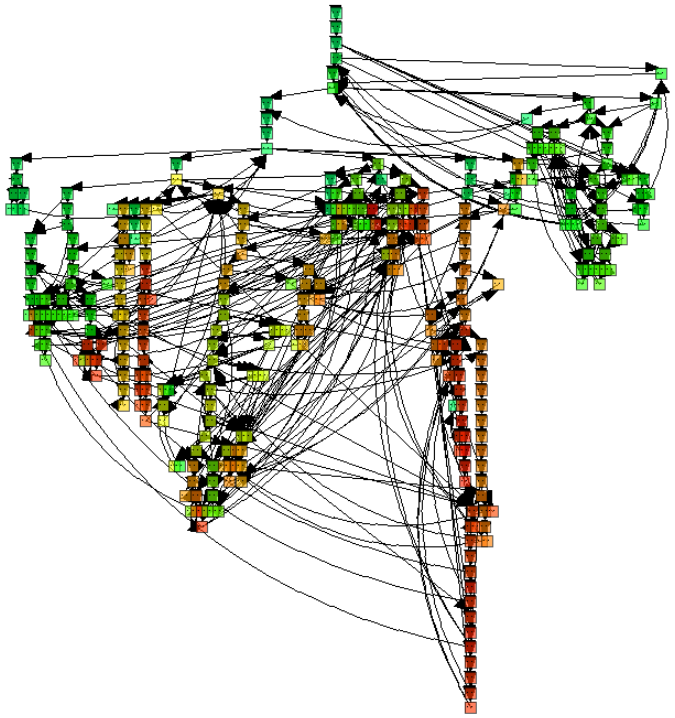

Figure 4.7: Our visualization for the graph for a text analysis program, showing the red section of a deeply nested loop, which contained a security vulnerability.

Fig. 4.9 illustrates the difference in the spanning trees and the vertex numberings obtained in our SFR search, and a more conventional depth-first search. These are obtained for a code segment containing a switch statement. The tree and the numbering obtained in the SFR shows the structure of the switch statement more naturally.

The SFR search tree and the SFR numbering also enables viewers to visually identify isomorphic subgraphs in the graph, which correspond to identical or equivalent lines of code in the program; see Fig. 4.10.

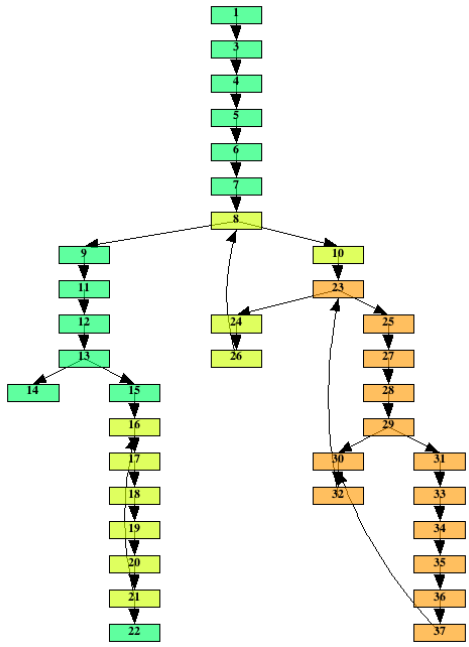

Figure 4.8: A graph for the bubblesort algorithm; vertices are labeled with SFR numbers.

4.6.2 Additional Figures and Examples

Here we show the graphs for some simple algorithms, as visualized in our J-viz.

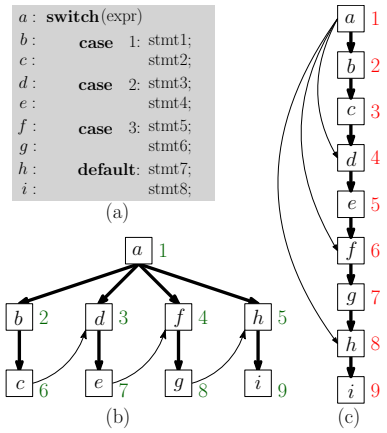

Figure 4.9: (a) A code segment containing a switch statement, (b) the spanning tree (thick edges) and vertex numbering (in blue color) for the graph obtained from SFR search, and (c) the spanning tree (thick edges) and vertex numbering (in red color) for the graph obtained from depth-first search.

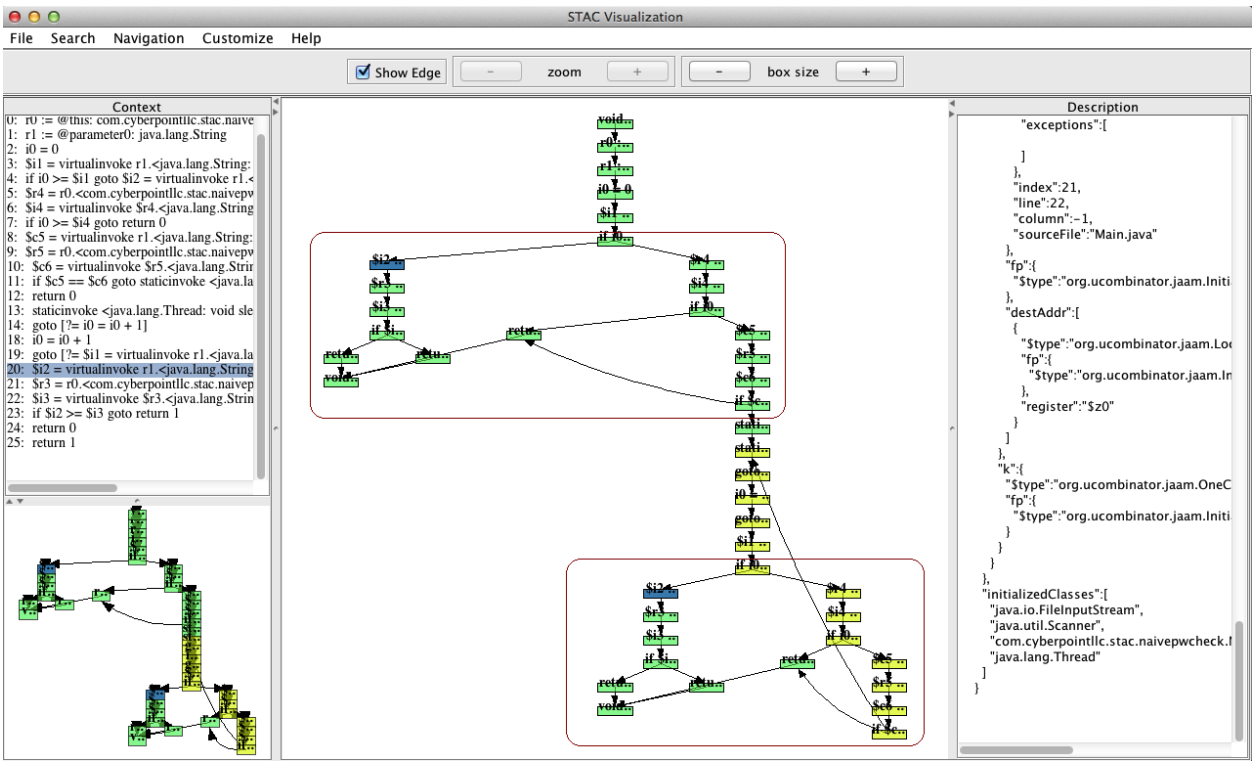

Figure 4.10: Visually separable isomorphic subgraphs in the graph for the password checker code; these isomorphic subgraphs correspond to the same or equivalent lines of code.

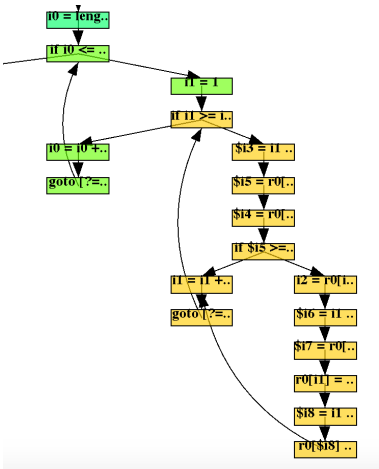

Figure 4.11: This partial graph of a bubblesort algorithm shows how drawing upward edges as curves and highlighting nested loops can improve readability. It also shows our use of colors for different levels of nested loops.

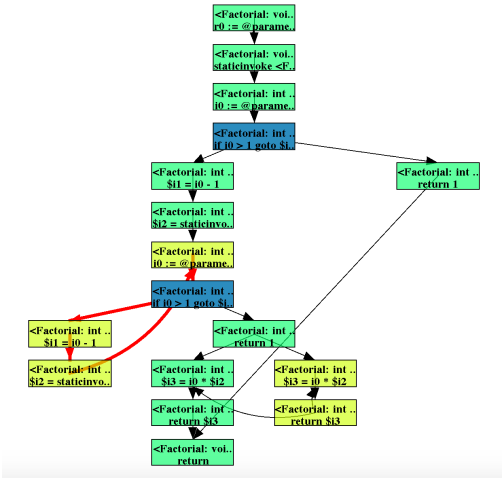

Figure 4.12: A graph of a recursive factorial function using 1-CFA. The highlighted instruction occurs twice - once in the context of being called from main, and once in the context of the factorial function calling itself. Note, in addition, how this tree is different from a breadth-first search (BFS) tree. Namely, there is a long forward edge from the rightmost node in the drawing. In a BFS numbering of this graph, that edge would force its end-vertex to a higher level, which would distort the natural notion of recursive depth that the SFR spanning tree illustrates better here.

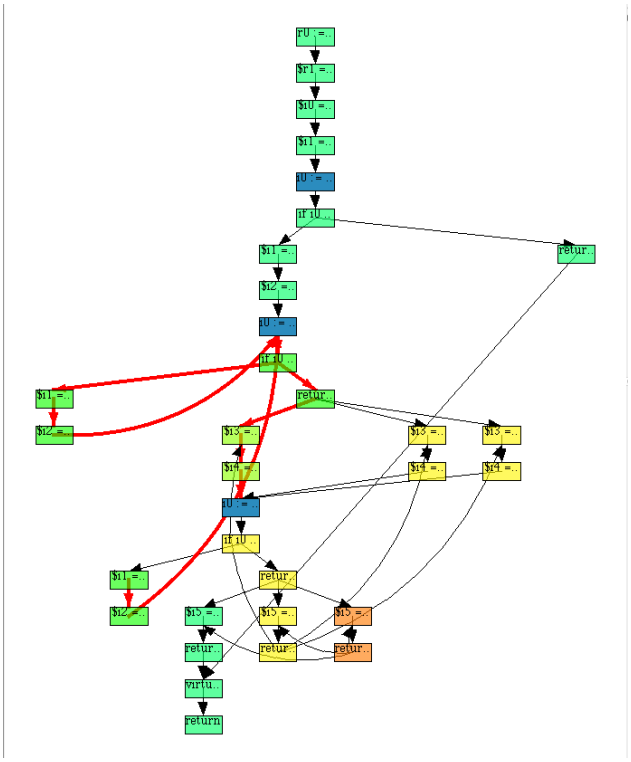

Figure 4.13: A graph of a recursive Fibonacci function using 1-CFA. The highlighted instruction occurs three times, because the function can be called from main, or from itself in two different places.

4.6.3 Illustration of the J-Viz System

Fig. 4.14 illustrates our J-viz system and its different component.

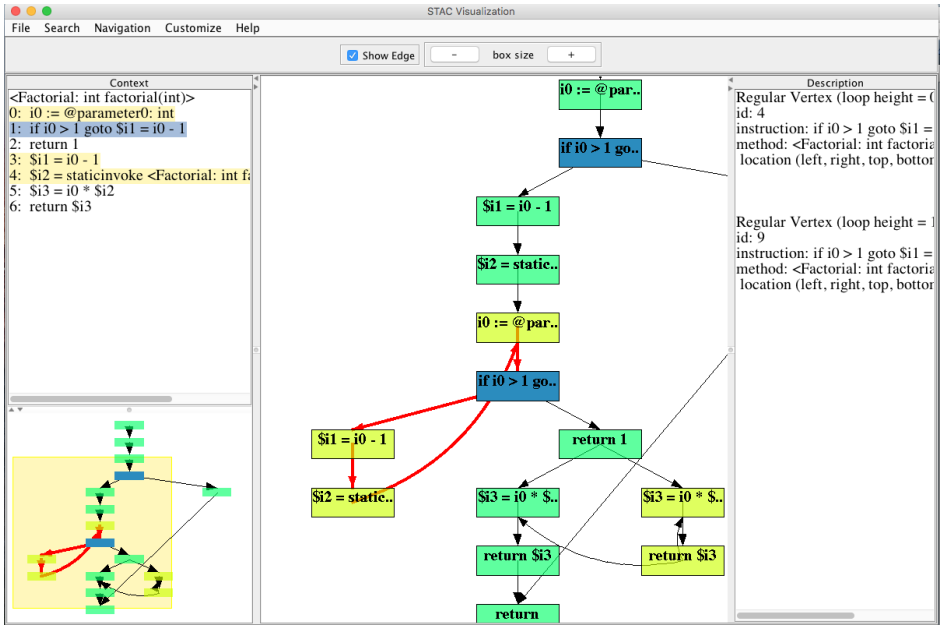

Figure 4.14: A screenshot of J-Viz with a (clipped) view of an example Factorial program. This shows the left panel with the code for the currently selected method, the right area with the description of each selected node, and the minimap with our current zoom level.

Fig. 4.15 illustrates how the J-viz system is used in identifying vulnerable code segments in a password checker program.

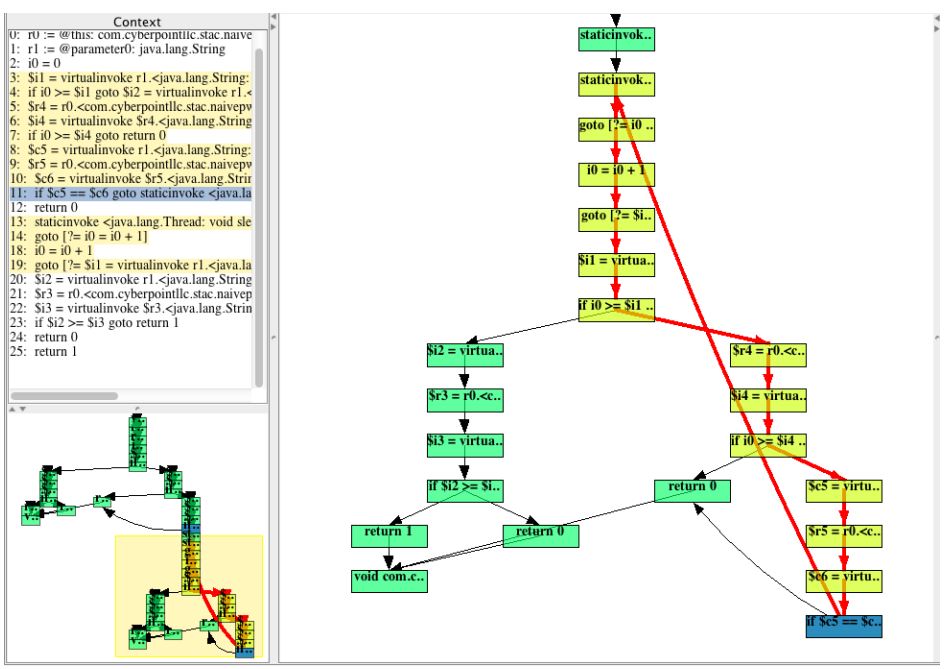

Figure 4.15: Our layout of the graph for a password checker with the relevant part zoomed, as a part of our first test case. The highlighted node shows a check for each character of a password. If this fails, the program exits the loop immediately, allowing for a side-channel attack (for identifying failed passwords).

4.7 Conclusion and Future work

We have described a new software visualization tool, J-Viz, that uses an SFR numbering scheme for graphs produced using the 1-CFA framework so that similar sections of code are drawn similarly and deeply nested code portions are placed well and highlighted.

In keeping with recent graph drawing research, we have argued that our system preserves the mental map of the user as they interact with the graph. We also meet all of the criteria that both programmers and researchers have considered essential for software visualization. As a result, our system has already proven to be useful for human analysts in finding various kinds of software vulnerabilities.

Although our tool is already fairly scalable, in future work, we plan to test our system for even larger programs. We also plan to study ways to provide semi-automated methods for identifying other kinds of potential algorithmic-complexity security vulnerabilities, such as those due to over-consumption of memory.

Bibliography

[1] M. J. Alam, D. Eppstein, M. Kaufmann, S. G. Kobourov, S. Pupyrev, A. Schulz, and T. Ueckerdt. Contact graphs of circular arcs. In *WADS '15*, volume 9214 of *LNCS*, pages 1–13. Springer, 2015.

[2] M. J. Alam, M. T. Goodrich, and T. Johnson. J-viz: Finding algorithmic complexity attacks via graph visualization of java bytecode. In *2016 IEEE Symposium on Visualization for Cyber Security (VizSec)*, pages 1–8, 2016.

[3] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. COSTA: Design and implementation of a cost and termination analyzer for java bytecode. In *Formal Methods for Components and Objects*, pages 113–132, 2007.

[4] P. Angelini, M. A. Bekos, T. Bruckdorfer, J. Hančl, M. Kaufmann, S. Kobourov, A. Symvonis, and P. Valtr. Low ply drawings of trees. In *International Symposium on Graph Drawing and Network Visualization*, pages 236–248. Springer, 2016.

[5] P. Angelini, S. Chaplick, F. De Luca, J. Fiala, J. Hancl Jr, N. Heinsohn, M. Kaufmann, S. Kobourov, J. Kratochvil, and P. Valtr. On vertex-and empty-ply proximity drawings. *arXiv preprint arXiv:1708.09233*, 2017.

[6] S. Bassi and R. K. Keller. Software visualization tools: Survey and analysis. In *Program Comprehension, 2001. IWPC 2001. Proceedings. 9th International Workshop on*, pages 7–17. IEEE, 2001.

[7] G. D. Battista, V. D. Donato, M. Patrignani, M. Pizzonia, V. Roselli, and R. Tamassia. Bitconeview: visualization of flows in the bitcoin transaction graph. In *IEEE Symp. on Visualization for Cyber Security (VizSec)*, pages 1–8, 2015.

[8] C. Bowen, S. Durocher, M. Löffler, A. Rounds, A. Schulz, and C. D. Tóth. Realization of simply connected polygonal linkages and recognition of unit disk contact trees. In *GD '15*, volume 9411 of *LNCS*, pages 447–459. Springer, 2015.

[9] H. Breu and D. G. Kirkpatrick. Unit disk graph recognition is NP-hard. *Computational Geometry*, 9(1-2):3–24, 1998.

[10] S. Bridgeman and R. Tamassia. Difference metrics for interactive orthogonal graph drawing algorithms. In S. H. Whitesides, editor, *6th Int. Symp. on Graph Drawing (GD)*, pages 57–71, 1998.

[11] S. Chaplick, S. G. Kobourov, and T. Ueckerdt. Equilateral l-contact graphs. In *WG '13*, volume 8165 of *LNCS*, pages 139–151. Springer, 2013.

[12] R. J. Cimikowski. Finding hamiltonian cycles in certain planar graphs. *Inf. Process. Lett.*, 35(5):249–254, 1990.

[13] U. Combinator. Jaam: JVM abstracting abstract machine. `https://github.com/Ucombinator/jaam`. Accessed 2016-06-10.

[14] G. Da Lozzo, W. E. Devanny, D. Eppstein, and T. Johnson. Square-contact representations of partial 2-trees and triconnected simply-nested graphs. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 92. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[15] F. De Luca, E. Di Giacomo, W. Didimo, S. Kobourov, and G. Liotta. An experimental study on the ply number of straight-line drawings. In *International Workshop on Algorithms and Computation (to appear)*. Springer, 2017.

[16] W. De Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. Vlissides, and J. Yang. Visualizing the execution of java programs. In *Software Visualization*, pages 151–162, 2002.

[17] R. DeLine, G. Venolia, and K. Rowan. Software development with code maps. *Commun. ACM*, 53(8):48–54, 2010.

[18] E. Di Giacomo, W. Didimo, S.-h. Hong, M. Kaufmann, S. G. Kobourov, G. Liotta, K. Misue, A. Symvonis, and H.-C. Yen. Low ply graph drawing. In *Information, Intelligence, Systems and Applications (IISA), 2015 6th International Conference on*, pages 1–6. IEEE, 2015.

[19] S. Diehl and C. Görg. Graphs, they are changing: Dynamic graph drawing for a sequence of graphs. In M. T. Goodrich and S. G. Kobourov, editors, *10th Int. Symp. on Graph Drawing (GD)*, pages 23–31, 2002.

[20] C. A. Duncan, D. Eppstein, M. T. Goodrich, S. G. Kobourov, and M. NÃűllenburg. Lombardi drawings of graphs. *Journal of Graph Algorithms and Applications*, 16(1):85–108, 2012.

[21] J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North, and G. Woodhull. Graphviz and dynagraph — static and dynamic graph drawing tools. In M. Jünger and P. Mutzel, editors, *Graph Drawing Software*, pages 127–148, 2004.

[22] D. Eppstein and M. T. Goodrich. Studying (non-planar) road networks through an algorithmic lens. In *Proceedings of the 16th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, page 16. ACM, 2008.

[23] C. Erten, P. J. Harding, S. G. Kobourov, K. Wampler, and G. Yee. Graphael: Graph animations with evolving layouts. In G. Liotta, editor, *11th Int. Symp. on Graph Drawing (GD)*, pages 98–110, 2004.

[24] K. Falconer. *Fractal Geometry: Mathematical Foundations and Applications.* John Wiley & Sons, 2004.

[25] S. Felsner and M. C. Francis. Contact representations of planar graphs with cubes. In *SoCG '11*, pages 315–320. ACM, 2011.

[26] T. Fraser. Space/time analysis for cybersecurity (STAC). `http://www.darpa.mil/program/space-time-analysis-for-cybersecurity`.

[27] F. Frati and M. Patrignani. A note on minimum-area straight-line drawings of planar graphs. In S.-H. Hong, T. Nishizeki, and W. Quan, editors, *15th Int. Symp. on Graph Drawing (GD)*, pages 339–344, 2008.

[28] E. R. Gansner, Y. Hu, and S. Kobourov. Gmap: Visualizing graphs and clusters as maps. In *Visualization Symposium (PacificVis), 2010 IEEE Pacific*, pages 201–208. IEEE, 2010.

[29] E. R. Gansner, Y. Koren, and S. North. Graph drawing by stress majorization. In *International Symposium on Graph Drawing, LNCS 3383*, pages 239–250. Springer, 2004.

[30] E. R. Gansner, E. Koutsofios, S. C. North, and G.-P. Vo. A technique for drawing directed graphs. *Software Engineering, IEEE Transactions on*, 19(3):214–230, 1993.

[31] A. Garg and R. Tamassia. Planar drawings and angular resolution: Algorithms and bounds. In J. van Leeuwen, editor, *2nd European Symp. on Algorithms (ESA)*, pages 12–23, 1994.

[32] D. Gonçalves, B. Lévêque, and A. Pinlou. Triangle contact representations and duality. *Discrete Comput. Geom.*, 48(1):239–254, 2012.

[33] M. T. Goodrich and T. Johnson. Low ply drawings of trees and 2-trees. In *Canadian Conference on Computational Geometry*, pages 2–10, 2018.

[34] M. T. Goodrich and P. Pszona. Streamed graph drawing and the file maintenance problem. In S. Wismath and A. Wolff, editors, *21st Int. Symp. on Graph Drawing (GD)*, pages 256–267, 2013.

[35] S. Hachul and M. Jünger. Drawing large graphs with a potential-field-based multilevel algorithm. In *International Symposium on Graph Drawing, LNCS 3383*, pages 285–295. Springer, 2004.

[36] H. Harborth. Lösung zu Problem 664A. *Elemente der Mathematik*, 29:14–15, 1974.

[37] P. Havlak. Nesting of reducible and irreducible loops. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(4):557–567, 1997.

[38] N. Heintze and D. McAllester. On the cubic bottleneck in subtyping and flow analysis. In *Logic in Computer Science, 1997. LICS'97. Proceedings., 12th Annual IEEE Symposium on*, pages 342–351. IEEE, 1997.

[39] P. Hliněný. Contact graphs of curves. In *International Symposium on Graph Drawing*, pages 312–323. Springer, 1995.

[40] P. Hliněný. Classes and recognition of curve contact graphs. *Journal of Combinatorial Theory, Series B*, 74(1):87–103, 1998.

[41] P. Hliněný. Contact graphs of line segments are NP-complete. *Discrete Math.*, 235(1-3):95–106, 2001.

[42] D. V. Horn and H. G. Mairson. Deciding $k$cfa is complete for EXPTIME. In *ACM SIGPLAN international conference on Functional programming, (ICFP'08)*, pages 275–282. ACM, 2008.

[43] D. V. Horn and M. Might. Abstracting abstract machines: a systematic approach to higher-order program analysis. *Communications of the ACM*, 54(9):101–109, 2011.

[44] T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31(1):7–15, 1989.

[45] F. Mansman, L. Meier, and D. A. Keim. Visualization of host behavior for network security. In J. R. Goodall, G. Conti, and K.-L. Ma, editors, *Proc. of the Workshop on Visualization for Computer Security (VisSec)*, pages 187–202, 2008.

[46] M. Might. *Environment Analysis of Higher-Order Languages.* PhD thesis, Georgia Institute of Technology, 2007.

[47] M. Might, Y. Smaragdakis, and D. V. Horn. Resolving and exploiting the k-cfa paradox: illuminating functional vs. object-oriented program analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, (PLDI'10)*, pages 305–315. ACM, 2010.

[48] G. L. Miller, S.-H. Teng, W. Thurston, and S. A. Vavasis. Geometric separators for finite-element meshes. *SIAM Journal on Scientific Computing*, 19(2):364–386, 1998.

[49] K. Misue, P. Eades, W. Lai, and K. Sugiyama. Layout adjustment and the mental map. *Journal of Visual Languages & Computing*, 6(2):183–210, 1995.

[50] D. Mondal and L. Nachmanson. A new approach to graphmaps, a system browsing large graphs as interactive maps. *arXiv preprint arXiv:1705.05479*, 2017.

[51] L. Nachmanson, R. Prutkin, B. Lee, N. H. Riche, A. E. Holroyd, and X. Chen. Graphmaps: Browsing large graphs as interactive maps. In *International Symposium on Graph Drawing and Network Visualization*, pages 3–15. Springer, 2015.

[52] H. Purchase. Which aesthetic has the greatest effect on human understanding? In *International Symposium on Graph Drawing*, pages 248–261. Springer, 1997.

[53] H. C. Purchase, E. Hoggan, and C. Görg. How important is the "mental map"? – an empirical investigation of a dynamic graph layout algorithm. In M. Kaufmann and D. Wagner, editors, *14th Int. Symp. on Graph Drawing (GD)*, pages 184–195, 2007.

[54] D. A. Quist and L. M. Liebrock. Visualizing compiled executables for malware analysis. In *6th Int. Workshop on Visualization for Cyber Security (VizSec)*, pages 27–32, 2009.

[55] O. Schramm. *Combinatorically Prescribed Packings and Applications to Conformal and Quasiconformal Maps*. PhD thesis, Princeton University, 1990.

[56] O. Schramm. Square tilings with prescribed combinatorics. *Israel J. Math.*, 84(1-2):97–118, 1993.

[57] O. G. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, 1991.

[58] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983.

[59] K. Stephenson. *Introduction to Circle Packing: The theory of discrete analytic functions*. Cambridge University Press (1), 2005.

[60] R. Tamassia, B. Palazzi, and C. Papamanthou. Graph drawing for security visualization. In I. G. Tollis and M. Patrignani, editors, *16th Int. Symp. on Graph Drawing (GD)*, pages 2–13, 2009.

[61] D. van Heesch. Doxygen: Source code documentation generator tool. *Available online: http://www.doxygen.org*, 2008.

[62] M. Wagner, F. Fischer, R. Luh, A. Haberson, A. Rind, D. A. Keim, and W. Aigner. A survey of visualization systems for malware analysis. In R. Borgo, editor, *Eurographics Conference on Visualization (EuroVis)*, pages 105–125, 2015.

[63] T. Wei, J. Mao, W. Zou, and Y. Chen. A new algorithm for identifying loops in decompilation. In *Static Analysis*, pages 170–183, 2007.

[64] Wikipedia contributors. Seven Bridges of Königsberg — Wikipedia, the free encyclopedia, 2018. Online; accessed 20-August-2018.