FORCE-DIRECTED GRAPH DRAWING AND AESTHETICS MEASUREMENT IN A NON-STRICT PURE FUNCTIONAL PROGRAMMING LANGUAGE

Christopher James Gaconnet

Thesis Prepared for the Degree of

MASTER OF SCIENCE

UNIVERSITY OF NORTH TEXAS

December 2009

APPROVED:

Paul Tarau, Major Professor
Philip H. Sweany, Committee Member
Roy T. Jacob, Committee Member
Ryan Garlick, Committee Member
Bill Buckles, Departmental Graduate Coordinator
Ian Parberry, Chair of the Department of Computer Science and Engineering
Michael Monticino, Dean of the Robert B. Toulouse School of Graduate Studies

Gaconnet, Christopher James. Force-Directed Graph Drawing and Aesthetics Measurement in a Non-Strict Pure Functional Programming Language. Master of Science (Computer Science), December 2009, 66 pp., 7 tables, 10 figures, references, 63 titles.

Non-strict pure functional programming often requires redesigning algorithms and data structures to work more effectively under new constraints of non-strict evaluation and immutable state. Graph drawing algorithms, while numerous and broadly studied, have no presence in the non-strict pure functional programming model. Additionally, there is currently no freely licensed standalone toolkit used to quantitatively analyze aesthetics of graph drawings. This thesis addresses two previously unexplored questions. Can a force-directed graph drawing algorithm be implemented in a non-strict functional language, such as Haskell, and still be practically usable? Can an easily extensible aesthetic measuring tool be implemented in a language such as Haskell and still be practically usable? The focus of the thesis is on implementing one of the simplest force-directed algorithms, that of Fruchterman and Reingold, and comparing its resulting aesthetics to those of a well-known C++ implementation of the same algorithm.

Copyright 2009

by

Christopher James Gaconnet

ACKNOWLEDGEMENTS

I am eternally grateful to my fiancee, Rachel, for supporting me on so many fronts while I increasingly neglected her to finish the thesis. Without her support my graduate school career may have continued indefinitely. Likewise, I both thank and apologize to my mom and sister for so kindly accepting long periods of neglect during recent months.

I express deep thanks to my major advisor, Dr. Paul Tarau, for always staying entirely supportive and for providing many enlightening conversations on computational topics. I have tremendous gratitude for Dr. Philip Sweany who provided much clear, intelligent advice and who so kindly suffered my prolixity through several email conversations. I thank Dr. Jacob and Dr. Garlick for serving on my committee and I am grateful to Dr. Garlick for providing a positive environment and useful advice ever since I first came to UNT as an undergraduate.

TABLE OF CONTENTS

Page

ACKNOWLEDGEMENTS iii

LIST OF TABLES vi

LIST OF FIGURES vii

Chapters

1. INTRODUCTION 1

2. THE GRAPH DRAWING PROBLEM 3

2.1 Representation 4

2.2 Technique 5

2.3 Aesthetic 6

3. FORCE-DIRECTED PLACEMENT 9

3.1 Concepts of Force-Directed Placement 9

3.2 Advantages of Force-Directed Placement 10

3.3 Disadvantages of Force-Directed Placement 11

3.4 Fruchterman and Reingold's Algorithm (FR91) 12

3.4.1 The Grid Variant 14

3.4.2 The 3D Variant 14

3.4.3 Remarks on FR91 15

4. NON-STRICT PURE FUNCTIONAL PROGRAMMING 16

4.1 Common Functional Abstractions 16

4.1.1 Strong Typing 17

4.1.2 Static Typing 18

4.1.3 Algebraic Data Types 19

4.1.4 Polymorphism 20

4.1.5 Non-strict Semantics 23

4.1.6 Immutable State 24

4.2 A Case for Graph Drawing in Haskell 26

5. FORCE-DIRECTED PLACEMENT AND AESTHETIC MEASUREMENT IN HASKELL ... 29

5.1 FR91 in Haskell 29

5.1.1 Random Node Placement 29

5.1.2 The Outer Loop 30

5.1.3 Computing Repulsive and Attractive Forces 32

5.1.4 Cooling and Bounding 34

5.2 Evaluating Functional FR91 36

5.2.1 Comparison of Results 36

5.2.2 Comparison of Running Times 46

5.3 A Freely Licensed Toolkit for Aesthetic Measurement of Graph Drawings48

5.3.1 Simple Aesthetics 48

5.3.2 Testing for Edge Crossing 50

5.3.3 A Simple Algorithm for Testing Approximate Geometric Parallelism 53

5.3.4 An Efficient Output-Sensitive Algorithm Counting Maximum Approximately Parallel Edges 55

6. CONCLUSIONS AND FUTURE WORK 58

6.1 Future Work 58

REFERENCES 60

LIST OF TABLES

Page

5.1 Crossing measures for HSFR 40

5.2 Crossing measures for BGLFR 41

5.3 Differences in crossing measures 44

5.4 Measures of normalized edge lengths.. 45

5.5 Difference in normalized edge length standard deviations 46

5.6 Difference in mean edge length 47

5.7 User time on Rome graphs 48

LIST OF FIGURES

Page

3.1 Nonplanar drawing of a planar graph 12

5.1 Similarities between HSFR and BGLFR 38

5.2 Differences between HSFR and BGLFR 39

5.3 Worst crossings for both implementations 41

5.4 BGLFR's largest excess of crossings 42

5.5 HSFR's largest excess of crossings 43

5.6 Most and least uniform edges for a BGLFR drawing 43

5.7 Most and least uniform edges for a HSFR drawing 45

5.8 Typical outcome of HSFR and BGLFR implementations 46

5.9 A drawing with aesthetic measures included in the image 51

CHAPTER 1

INTRODUCTION

Graph drawing algorithms, while numerous and broadly studied, have no presence in the non-strict pure functional programming model. The majority of graph drawing algorithms are implemented in C, C++, or Java. Although differences among programming languages are often syntactic and rarely instigate conceptual changes to algorithms, some languages do require taking special precautions to implement an algorithm under its original time and space complexity bounds.

Non-strict pure functional programming often requires redesigning algorithms and data structures to work more effectively under new constraints of non-strict evaluation and immutable state. Non-strict semantics provide powerful avenues for recursion which are impossible under a strict programming model; however, such semantics can easily lead to space leaks in an implementation. Strong typing and immutable state introduce new complexities of their own, such as the lack of null references and an inability to easily and efficiently update fields in a data structure.

This thesis addresses two previously unexplored questions:

- Can a force-directed graph drawing algorithm be implemented in a non-strict functional language, such as Haskell, and still be practically usable?
- Can an easily extensible aesthetic measuring tool be implemented in a language such as Haskell and still be practically usable?

The original contributions include a non-strict, pure functional implementation of the Fruchterman and Reingold [26] force-directed graph drawing algorithm along with some functional implementations of graph aesthetic measuring algorithms. I provide aesthetic measurements to establish that the functional implementation exhibits reasonably correct behavior in comparison to a popular C++ implementation of the same algorithm, and I briefly analyze user running time of the functional implementation and outline the steps necessary to improve performance. I introduce an unusual graph aesthetic, the number of approximately parallel lines in a drawing, and provide a division-free algorithm for testing approximate parallelism. I also sketch an output-sensitive algorithm for counting the maximum number of (not necessarily pairwise) approximately parallel edges in a drawing. The algorithm makes use of an interval map data structure for its efficient operation. For sparse graphs the algorithm is faster than naive checking of all edge pairs, but for dense graphs the running time is worse than the naive approach.

Graph drawing has almost no presence in the functional language Haskell (Section 4.2). More generally, graph algorithms are rather underrepresented in non-strict pure functional programming. As a result, the most significant contribution of this thesis is establishment of a starting point for further adaptation of graph drawing and aesthetic measurement algorithms to a non-strict pure functional environment, Haskell.

The rest of the thesis is structured as follows: Chapter 2 begins with a formal definition of 2D straight-line graph drawing and surveys a sample of graph drawing background. Chapter 3 describes force-directed drawing algorithms with details of the specific algorithm implemented in this thesis. Chapter 4 surveys the primary abstractions in non-strict, pure functional programming and motivates the problem of graph drawing in Haskell. The original contributions from this thesis are explained in Chapter 5, and finally, Chapter 6 wraps up the thesis with conclusions and future work.

CHAPTER 2

THE GRAPH DRAWING PROBLEM

Let $G = (V, E)$ be a finite graph with vertices V and edges $E \subseteq V \times V$. Authors commonly denote $|V|$ as n and $|E|$ as m. A *2D drawing* of G (from now on simply denoted a *drawing*) is an embedding of the vertices as points in 2-dimensional Euclidean space and of the edges as plane curves in that same space. The *graph drawing problem* is to take the abstract graph G and produce a drawing of G. Since loops, parallel edges, and multiple components add little if any complexity to theoretical results, most discussions of graph drawing assume simple, connected graphs. This thesis makes the same assumption.

For any graph G, there exists an infinite number of ways to draw G in the plane. Furthermore, there are multitudes of alternative definitions for graph drawing which do not necessarily involve the Euclidean plane or representing vertices as points or edges as curves. Such diversity results in a rich, extensive history within graph drawing. This history contains a vast range of techniques which produce a vast assortment of visual representations.

The rest of this section surveys portions of graph drawing's history as background and related work to the specific techniques addressed in the following chapters. More extensive backgrounds can be found in surveys [2, 13, 33] and textbooks [17, 57, 58]. A list of open problems in graph drawing is available in [10].

In the context of this thesis, we can think of graph drawing as composed of three primary subtopics: *representation*, which describes how a graph is visualized; *technique*, which describes how an abstract graph is transformed into a representation; and *aesthetic*, which provides a foundation for quantitative measure of qualitative properties.

2.1. Representation

Among the many possible representations for graphs, this thesis focuses on *straight-line drawings*, where edges are drawn as straight line segments between their corresponding vertices. Straight-line drawings are one instance of *polyline drawings*—drawings where each edge is a polygonal chain. Other polyline representations exist. In an *orthogonal drawing* each edge is a chain drawn exclusively as horizontal and vertical line segments. *Grid drawings* have integer coordinates for all vertices and edge bends. *Radial drawings* place vertices on concentric circles and provide a useful way to visualize *free trees*, trees with no root. The polylines in a polyline drawing can be generalized to plane curves resulting in a *curved-line drawing*.

Planar drawings have no two edges intersecting except at their incident vertices. Graphs which can be drawn as planar drawings are called *planar graphs*. Planar graphs can be characterized without reference to graph drawing using concepts of either *forbidden graphs* or *forbidden minors*. Avoiding the need to compute and check a potentially infinite number of drawings, these combinatorial characterizations have led to linear time algorithms for testing planarity and for identifying the nonplanar forbidden graphs, or *Kuratowski subgraphs*, in any nonplanar graph [7, 8, 16, 32, 35, 54]. Because drawings of planar graphs do not have edge intersections, or *crossings*, they play a major role in graph drawing.

Delta-confluent drawings [22] simplify edge crossings by merging crossing edges together in a manner resembling train tracks. More exotically, graphs drawn onto hyperbolic geometry [45] provide a “fish-eye” view of complicated data. *Visibility representation* draws vertices as horizontal lines and edges as vertical lines. The vertical lines intersect exactly the horizontal lines that represent incident vertices. Such a representation looks nothing like the typical drawing of a graph, yet it presents useful information in certain contexts and has applications in integrated circuit design.

This brief sampling of the variety of representations bears mentioning because—as described in Section 4.2—only rooted tree drawings currently have any presence in non-strict, pure functional programming.

2.2. Technique

Each of the above representations requires some algorithm to transform an abstract graph into a corresponding drawing. Some representations, such as straight-line, orthogonal, and grid, have multiple algorithms which can yield an appropriate representation. Hence, however many diverse representations are known, there is an even larger number of techniques to yield them.

This thesis focuses on one of the simpler techniques, *force-directed placement*, which turns an abstract graph into a physical simulation to create a straight-line drawing. It is described in more detail in Chapter 3.

Other techniques exist which take completely different approaches to graph drawing. Like force-directed layout some use heuristics while others take exact graph theoretic approaches.

A *planar embedding* is a planar graph together with a clockwise ordering of the edges around the vertices. Most planarity testing algorithms provide a planar embedding for any graph they certify as planar. A planar straight-line embedding can then be drawn on a grid as a planar drawing in linear time and in $O(n^2)$ area [52, 58].

Since drawing planar graphs is efficient and relatively simple, one can attempt to draw a nonplanar graph by *planarizing* the graph, drawing it, and then drawing any remaining nonplanar edges. This might involve finding a *maximum planar subgraph* and then adding the remaining edges onto the planar drawing; however, the maximum planar subgraph problem is NP-hard [29]. Using *maximal planar subgraphs* provides a convenient approximation with $O(n + m)$ time complexity. Yet another approach to planarization is isolating Kuratowski subgraphs and placing placeholder vertices on their crossings.

Directed graphs can be drawn in a hierarchical manner by placing the vertices on levels according to their topological order. Two vertices are on the same level if and only if their shortest paths from some chosen root have equal lengths. After placing the vertices on the correct levels, hierarchical drawing algorithms then search for permutations of the vertices on each level which minimize edge crossings. Finding optimal permutations even for only two levels is NP-hard, so practical algorithms use heuristics. Once the permutations have been chosen, these algorithms can optionally adjust the positions of vertices on each level (such that permutations are not changed) to minimize the number of polyline edge bends.

As with the representations, this brief sampling of the variety of techniques available for graph drawing bears mentioning because only the hierarchical directed tree technique currently has any presence in non-strict pure functional programming.

2.3. Aesthetic

Since transforming a graph into a visual representation is rarely trivial, it's useful to have a way to measure whether a resulting drawing matches a human's expectations for the intended representation. *Aesthetics* are empirical properties of drawings which provide such measure. Numeric measure of aesthetic value goes at least as far back as Birkhoff in *Aesthetic Measure* [6]. Some example aesthetics for graph drawings include number of edge crossings, symmetry, and angular resolution. The usefulness of an aesthetic depends on the intended purpose of a visual representation.

Some combinations of aesthetics are *competitive*. For example, there exist planar graphs for which minimizing edge lengths results in nonplanar drawings, i.e. you cannot always optimally minimize edge lengths and crossings together. This example has particular relevance since a study by Purchase [49] indicates that the number of crossings may be the most significant aesthetic for human understanding in graph drawings. Even so, the intended purpose of a graph drawing plays a key role in determining the significance of any particular

aesthetic. A more recent study has shown that for the task of finding shortest paths, keeping multi-edge paths as straight as possible can have more significance on human performance than reducing crossings on the paths [63]. As stated, "the cognitive cost of a single crossing is approximately the same as 38 degrees of continuity." Why any human would take on the task of finding shortest paths is another matter entirely.

The above multidisciplinary studies which cross into the fields of cognitive psychology and information visualization are notable in that they are providing an empirical foundation for evaluation of graph drawing algorithms. This foundation for measure is necessary for any field to become mature; hence, implementors of graph drawing algorithms should subject their implementations to these measures on large test samples rather than relying solely on subjective analysis. As discussed in Chapter 5, part of the work of this thesis is to provide an easily extensible framework for performing aesthetic measurements.

The following list describes some common aesthetics:

- *Angular extrema* Minimum and maximum angle between two adjacent edges
- *Area* Area of the smallest rectangle bounding the drawing
- *Crossings* Total number of edge crossings
- *Bend extrema* Minimum and maximum number of bends in an edge in a polyline drawing
- *Bends* Total number of edge bends in a polyline drawing
- *Edge length extrema* Minimum and maximum edge lengths, possibly with max edge length normalized to 1
- *Symmetry* Reflection and rotational symmetry in a drawing
- *Uniform bends* Standard deviation of the number of bends per edge in a polyline drawing
- *Uniform edge length* Standard deviation of edge lengths

- *Uniform vertex placement* Standard deviation of distances between each vertex and the geometrically nearest vertex

As a final remark on aesthetic, note that aesthetics have use beyond quantitative analysis. In several algorithms, including the force-directed algorithm of Davidson and Harel [15], aesthetics are used as parameters to the algorithm, letting a user tune the behavior of the algorithm by prioritizing different aesthetics.

CHAPTER 3

FORCE-DIRECTED PLACEMENT

Force-directed placement was formally introduced by Eades in [20], though the practice goes at least as far back as Tutte's barycentric method [60] and engineering applications in VLSI [51]. The term *spring-embedder algorithm*, coined by Eades, is often used interchangeably with force-directed placement.

In force-directed placement, vertices, edges, and at times graph theoretic properties are modeled as physical entities exerting "forces" on one another. The scare quotes on "forces" indicate that these are not always modeled using the typical physicist's definition of *force*—that is, the forces rarely represent a proportionality of mass and acceleration.

Instead, "forces" in force-directed placement often directly represent a vector displacement, acting as a velocity applied to a body with an assumed time step of 1 unit. Authors accept such a break from the formal physical definition since it often simplifies models and yields practical results.

In a way, real force, as the derivative of momentum, provides too powerful a model for graph drawing since it often leads to *dynamic equilibria* such as orbits and oscillations—a phenomenon easily witnessed by looking to our remarkably stable position in the universe. From this point on, I will leave the scare quotes off of "forces" with the intention of using the more loosely defined graph drawing definition of the word.

3.1. Concepts of Force-Directed Placement

In force-directed placement, vertices often behave like electromagnetic particles, such that they repel each other. Edges resemble springs between adjacent vertices which attract those vertices toward each other. This behavior proceeds in a physical simulation that runs until

some predetermined halting point is reached. Example halting points include reaching a set number of iterations or going into some minimized energy state.

To rephrase it imperatively, the high-level workings of a force-directed algorithm generally proceed as follows:

(i) Assign the vertices an initial position.

(ii) Compute repulsive forces between all pairs of vertices.

(iii) Compute attractive forces between all vertices with edges $(a, b) \in E$.

(iv) Sum corresponding repulsive and attractive forces.

(v) Update vertex positions using computed forces.

(vi) Repeat from Step (ii) until reaching fixed iteration count or some minimized energy state.

3.2. Advantages of Force-Directed Placement

An obvious advantage of force-directed placement is that it provides a simple, straightforward way to visualize graphs. Most variations on the approach do not rely on deep combinatorial results in graph theory. Due to this simplicity, it was among the first techniques for drawing general undirected graphs in practice. The general algorithm is simple to modify and experiment with, leading to wide popularity for this approach.

Force-directed placement is good at displaying symmetries of graphs. This was commonly known as folklore since their initial use, but it has been proven with theoretical results. Lin and Eades [21] proved that spring-embedder algorithms can always display graph symmetries via symmetries of a drawing as long as the algorithm fits into their general spring system. Some of the algorithms this applies to include Becker and Hotz [3], Eades [20], Fruchterman and Reingold [26] (with a clever trick), Kamada and Kawai [39], and Tutte [60].

By design, spring-embedder algorithms optimize for uniform edge length. This is obvious since each spring has minimal energy at its natural resting length. These algorithms are

also easily modified to optimize for uniform vertex placement by adding an "ideal vertex separation" constant to proportion the forces, such as in Fruchterman and Reingold [26].

With some modifications, force-directed algorithms have been shown to quickly draw graphs with up to 10^5 vertices [50, 28]. They have also been extended into higher dimensions [27] and non-Euclidean geometries [42].

3.3. Disadvantages of Force-Directed Placement

Force-directed algorithms do have their drawbacks. For general force-directed placement as described above, all graphs require $\Omega(n^2)$ time to draw. This bound can be improved with the use of geometric tricks such as spatial indexing, but appropriate structures must be chosen carefully with respect to the geometric assumptions that can be made about vertex and edge placement. Additionally, such techniques reduce the simplicity of the approach, which is its primary advantage. Depending on the application, choosing a linear planarity embedding algorithm or some other approach could be more appropriate in terms of computational efficiency.

Force-directed algorithms sometimes sacrifice crossing minimization for the sake of symmetry. A simple example is the complete graph K_4, which most force-directed algorithms draw with 1 crossing even though the graph is planar. To combat this, Bertault [4] and Dwyer, Marriott, and Wybrow [18, 19] have developed force-directed algorithms which, given an initial drawing, create a new drawing such that no new crossings are introduced. This technique has been useful in improving symmetry and other aesthetics in the output of graph-theoretic planar drawing algorithms.

Overall, the lack of graph-theoretic techniques in force-directed placement can be seen as a disadvantage. As with the planar example above, often times a graph has properties which could aid in the drawing of the graph. There exists much potential for experiments combining graph-theoretic and force-directed methods.

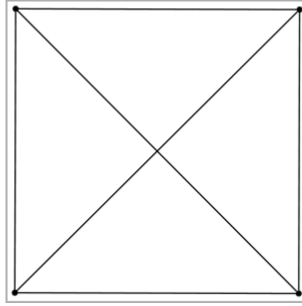

Figure 3.1. Nonplanar drawing of a planar graph.
K_4, a planar graph, drawn non-planar by my implementation of the Fruchterman and Reingold algorithm (Chapter 5). Many force-directed algorithms draw K_4 similarly.

Despite the above problems, the simplicity and effectiveness of force-directed placement leads to the conclusion that such algorithms provide a good starting point for studying graph drawing in a different model of computation. In this thesis, I begin this process with a particularly simple instance, the Fruchterman and Reingold 1991 algorithm [26].

3.4. Fruchterman and Reingold's Algorithm (FR91)

The algorithm of Fruchterman and Reignold [26] (from now on denoted *FR91*) generalized the work of Eades [20] with several improvements.

First, the algorithm integrates *uniform vertex placement* into the aesthetics it optimizes. It does this by creating an *ideal vertex separation* constant:

$$k = \sqrt{\frac{\text{area of frame}}{|V|}} \tag{1}$$

The constant k then directly proportions the attractive and repulsive forces, which are quadratic:

$$f_a(d) = \frac{d^2}{k} \tag{2}$$

$$f_r(d) = \frac{-k^2}{d} \tag{3}$$

In the above equations, d is a scalar distance between two vertices. These functions are defined to return scalar values. The algorithm logic is responsible for projecting the scalars onto vectors with appropriate dimension.

FR91 works just like the simplest of the force-directed algorithms. There is a primary iteration loop which will simply quit after a set number of iterations (typically 50). Within this loop are three inner loops. The first iterates through all pairs of vertices, computing f_r for each vertex in the pair. The second iterates through all edges, computing f_a for each vertex in the pairs $e = (a, b) \in E$. The third loop iterates through all vertices, limiting the max displacement to the current "cooling temperature," updating the vertex positions, and then limiting the max applied displacement to within the confines of the drawing frame. The cooling temperature is then updated for the next iteration and the process repeats.

The "cooling temperature" mentioned in the previous paragraph is an analogue to cooling in simulated annealing methods. As time goes on (modeled by iterations), the maximum possible displacement for the vertices is limited according to the cooling function. Its purpose is to escape dynamic equilibria and improve the likelihood of producing a quality drawing with quick termination. Fruchterman and Reingold experimentally explored several possibilities for cooling functions, but for simplicity suggest the function start at some percantage of the frame width and then decrease in an inversely linear proportion:

$$cool(t) = \frac{\text{width}}{t} \tag{4}$$

The authors punt on formal analysis of the outer loop of the algorithm, in regards to how many iterations are *required* to produce a satisfying drawing. Formal methods for reasoning about the behavior of force-directed algorithms were largely unknown at the time. Instead, the authors chose to iterate 50 times for the standard version of the algorithm and 70 times

for a grid variant to be discussed next. Both of those times are an improvement on Eades' 100 iterations [20].

3.4.1. The Grid Variant

Fruchterman and Reingold tried several alterations to the algorithm which provide tremendous insight into the inherent flexibility of force-directed placement. The first such experiment is a modification of the algorithm to divide the frame into square grids so that repulsive forces are only applied to geometrically close vertices in the current grid cell. However, the square shape of the grid cell imposes distortion on the vertex placement, so the repulsive force actually acts as if bound inside a circle inscribed in the grid cell. The grid approach brings some unexpected advantages such as allowing the algorithm to correctly draw disconnected graphs without special modifications (normally the components would repel each other into the frame boundaries). The grid also reduces the edge stretching effect of large cliques separated by sparse cuts. Repulsive forces under the grid variant are computed as follows:

$$f_r(d) = \frac{k^2}{d} u(2k - d) \tag{5}$$

$$u(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} \tag{6}$$

3.4.2. The 3D Variant

The other major experiment was an extension into 3-dimensional placement. For the FR91 algorithm, drawings that did not already look 3D when drawn in 2D turned out poorly when extended to 3D. Graphs resembling or directly constructed from 3D shapes were drawn well when extended into 3D.

3.4.3. Remarks on FR91

A notable aspect of [26] is that the entirety of the analysis is subjective. There are 83 labeled figures in the whole paper. At the time FR91 was invented, experimental analysis of graph drawings was not common practice. Although quantitative analysis of graph drawings has gained popularity since then, there still seems to be no unifying tool for aesthetic measurement. Aesthetic measuring algorithms are split among the many graph drawing frameworks, most likely reinvented several times.

Overall, FR91 works well for highly symmetric graphs, but it often draws planar graphs with crossings. Even so, its simplicity provides broad potential for extensibility, such as with spatial indexing or higher dimensions. It served as a starting point for many variations of spring-embedder algorithms that followed. It therefore serves as an excellent foundation for bringing graph drawing algorithms into a different programming model.

CHAPTER 4

NON-STRICT PURE FUNCTIONAL PROGRAMMING

In this chapter I highlight the primary methods of abstraction offered by functional programming languages (specifically Haskell). The intention is to motivate the idea that these abstractions offer potentially unexplored perspectives for graph drawing algorithms.

Currently the most popular graph drawing applications use the most popular imperative languages: C, C++, and Java. Here is a list of some of those applications with their primary implementation languages in parentheses: Boost Graph Library (C++), GraphEd (C), Graphviz (C), JGraph (Java), LEDA (C++), Pigale (C++), and Walrus (Java).

Traditionally, computer scientists maintain that implementation details, such as choice of programming language, have no relevance to concepts. We generally accept that algorithms and their properties are isomorphic across various programming languages. This is largely correct; the differences between an FR91 force-directed implementation in C and another in Java are likely to be mainly syntactic, with few conceptual changes if any.

On the other hand, the primary purpose of programming languages is to offer multiple abstractions. A large enough difference in the set of abstractions chosen for two different languages can force a dramatic change in fundamental concepts used when designing algorithms. The Haskell programming language arguably has such a set of abstractions when contrasted with C, C++, and Java. Under this presumption, Haskell has potential for offering new developments in the wide field of graph drawing algorithms.

4.1. Common Functional Abstractions

The following abstractions form the primary characteristics of the non-strict, pure functional programming language Haskell. This list is by no means exhaustive. Reputable books

describing Haskell and these concepts in more detail are becoming increasingly available [46, 47]. Haskell's distinguishing features are described in more depth and with historical context in [36].

4.1.1. Strong Typing

Strong typing, a moderately vague term, generally guarantees that certain kinds of errors will not happen among values of different types. This nebulous concept is best illustrated with examples. In C, you can cast an integer value into a pointer and then attempt to dereference that memory address. If the program does not have permission to access that memory or if the address is otherwise invalid (e.g. the pointer is dereferenced as a function call but the value at the address is not a function), a segmentation fault error occurs. In C you can also add the value of an integer variable to the value of a float variable and the integer is automatically *coerced* into a float. Likewise, in other languages, such as Perl, you can perform addition on differently typed values with automatic (sometimes tricky) coercion:

```
$ perl -e 'print 017+1;'
16
$ perl -e 'print "017"+1;'
18
```

In a typical strongly-typed language, such as Haskell, such casts are forbidden and coercions must be specified explicitly in code. Compilers are required to reject any program that is *ill-typed*, resulting in a guarantee that certain kinds of type-based errors will never appear in a running program. This has obvious benefits, but can also be inconvenient at times.

In regards to graph drawing, one inconvenience of strong typing is that many graph algorithms and data structures assume the ability to use and manipulate memory pointers. For example, the FR91 algorithm loops through all vertex pairs and then loops through all edges. Imagine a C implementation where the edges are implemented as pairs of integers

representing indices into a vertex array—a representation which does not require pointers. Now suppose the algorithm is used to animate a process of repeated vertex removals from a graph. Each time a vertex is deleted, the graph data structure must move at least one vertex in the vertex array; otherwise, the vertices are not contiguous and can no longer be iterated over with an integer-based loop. The edges which referenced the moved vertices must then be updated, so removing a vertex could become a $O(m)$ operation. A simple solution to avoid this overhead is to use pointers in the edges so that the edges do not need to be updated or to use pointers in the node array so that holes in the array can be easily detected as null pointers.

References, such those in Java, provide the benefits of pointers with a strong degree of type safety. Java, however, does allow null references, taking some of the "strength" out of the strong typing. Specifically, programs which dereference null references compile as perfectly valid Java programs.

Haskell, on the other hand, provides a stronger variant of strong typing, wherein null references are not allowed. Extrapolating from the above example of efficient vertex deletion, it becomes obvious that the core concepts of some algorithms and data structures must be changed if they are to be implemented with the same complexity bounds with which they were designed.

4.1.2. Static Typing

With static typing the compiler knows the type of every value and expression at compile time. This concept has close ties to strong typing; when static and strong typing are combined there can be no type errors at runtime (to the extent that the strong typing requires).

Static typing is often seen as tedious for the programmer since in statically typed languages such as C, C++, and Java, types must be explicitly annotated. Several functional languages, including Haskell, are notable in that they are statically typed but the programmer does

not have to annotate the code with types. The compiler can *infer* the types of most valid programs; though some valid programs are *ambiguous* in their possible types and must be annotated.

Static typing has no obvious effects on the concepts of graph drawing algorithms, but it bears mentioning for its close relationship with strong typing.

4.1.3. Algebraic Data Types

Algebraic data types define types by specifying their shape. Values get wrapped in data constructors and are unwrapped with a technique called *pattern matching*.

```
data Vec2 = Vec2 Double Double
fmap (Vec2 x y) = Vec2 (f x) (f y)
```

Listing 4.1. Algebraic data types, data constructors, and pattern matching.

Pattern matching helps reduce nesting and conditional syntax by often substituting conditionals with functions defined in parts.

```
fmap [] = []
fmap (x:xs) = f x:fmap f xs
```

Listing 4.2. Pattern matching replaces conditional statements as syntactic sugar.

Algebraic data types appear to only have syntactic effects on graph drawing algorithms until we look deeper into their implications. Haskell allows recursive types, meaning a type can contain values of its own type. The canonical example is the list, which is either the empty list or an item attached to the front of a list:

```
data List a = Empty | Attach a (List a)
```

Listing 4.3. Recursive types.

This type of recursive definition is called an *inductive* definition. It is the preferred way to define dynamic data structures in Haskell. Another example is a special type of graph, the binary ordered tree:

```
data Tree a = Leaf | Branch (Tree a) a (Tree a)
```

Listing 4.4. The binary ordered tree. All edges are directed toward child nodes.

General graphs can have cycles, loops, and parallel edges. Expanding the above tree definition to work for general graphs is non-trivial. Graphs are also dynamic, meaning it's common to add and remove vertices and edges. Since dynamic data structures are ideally defined inductively in functional programming, using the typical implementation of graphs as arrays or lists of vertices and edges is largely non-functional. Erwig has proposed an inductive definition for graphs [23]. His inductive definition is used in the implementations described in Chapter 5.

As a final remark on algebraic data types, Haskell further allows recursive higher-order types. Among other things this allows definitions of types as fixed points of types, as shown by Jones [37]:

```
data Mu f = In (f (Mu f))
data NatFix s = Zero | Succ s
data Nat = Mu NatFix
```

Listing 4.5. Recursive, higher-order types.

4.1.4. Polymorphism

Haskell supports two types of polymorphism: *parametric* and *ad-hoc*.

4.1.4.1. Parametric Polymorphism

Parametric polymorphism allows sharing of the shape of a data structure or the implementation of a function across a set of types. The list is the canonical example for data

structures. All lists, regardless of what type of value they contain, have a basic underlying linear structure: A list containing values of some type *a* looks like either the empty list or like a value of type *a* attached to the front of a list containing values of type *a*. Here, the type *a* is the *parameter* that varies across lists of different type. With parametric polymorphism only one generic definition of lists is required (`List a`) to support lists of ints (`List Int`), lists of strings (`List String`), or lists of any other type. The same concept applies to labeled vertices and edges in graphs which might need to contain labels of ints, vectors, strings, or any other type.

The canonical example of parametric polymorphism in a function is the function that reverses lists. The function does not depend on what type of value the list contains, so all list types can share the same implementation.

```
reverse :: List a -> List a
reverse Empty = Empty
reverse (Attach x xs) = reverse xs ++ x
```

Listing 4.6. Parametric polymorphism viewed from the perspective of polymorphic functions.

On graphs, parametric polymorphism is useful for generic graph operations such as reversing edge directions or computing depth-first search trees.

In C, parametric polymorphism would be accomplished with pointers; in C++ with pointers or templates; and in Java with generics or references and type casting. The notable aspect of Haskell's parametric polymorphism with respect to these languages is that it requires almost no syntactic overhead. Additionally, it has the full benefits of strong typing. We are not considering syntactic convenience as motivation for graph drawing in Haskell, but the feature bears mentioning because it is a fundamental form of abstraction.

4.1.4.2. Ad-Hoc Polymorphism via Type Classes

Ad-hoc polymorphism is a technique for *overloading*, sharing a name among differing implementations. Whereas parametric polymorphism shares a single implementation among multiple types, ad-hoc polymorphism shares a single name among multiple implementations and types.

Consider FR91, the Fruchterman and Reingold algorithm, implemented for both two and three dimensions. This implementation will have a function to compute the Euclidean distance between two vertices. In C, the 2D and 3D distance functions must have distinct names because C does not provide ad-hoc polymorphism. In C++ or Java, one could define a *base class* or *interface* which provides a public `distance` method that gets implemented appropriately for inherited vector subclasses of various dimensions. C++ and Java therefore support ad-hoc polymorphism via subclassing.

Haskell supports ad-hoc polymorphism via type classes. The primary distinction between type classes and object-oriented classes is the method of *dictionary passing* [36]. Although the distinction from object-oriented classes at first appears superficial, [36] cites several unexpected uses for type classes beyond simple overloading. The distinction becomes especially apparent when considering extensions such as multi-parameter type classes [38] and type families [53].

The immediate relevance of type classes to graph algorithms rises from assuming an inductive definition for graphs. Ongoing work in the Haskell standard libraries attempts to study operations over inductive and coinductive data with the purpose of finding generalizations that abstract into standard type classes. Such standard classes are well studied over simple structures such as lists and trees, but no published work has done the same for graphs.

4.1.5. Non-strict Semantics

Haskell has a non-strict semantics which most compilers implement via lazy evaluation. In Haskell's semantics, $\perp$ (pronounced *bottom*) is a value representing a term which has no normal form, also known as a *divergent term*. In practice it is used to represent an error or a computation which never halts. Pope provides an explanation in [48]:

> A function f is strict iff:
>
> $$f \perp = \perp$$
>
> That is, given a divergent argument, the function yields a divergent result. The same idea can be extended to functions of multiple arguments, where we would say "f is strict in its nth argument ..."

Pope continues by describing that programming languages with strict semantics use an evaluation order that requires every function to be strict in all its arguments and that *call-by-value* is the most common strict evaluation order.

Haskell's non-strict semantics and its relationship to lazy evaluation are further described by Pope:

> The converse of strict is "non-strict". Non-strict languages permit semantic equations like this (but strict languages do not):
>
> $$g \perp \neq \perp$$
>
> ...
>
> Call-by-name evaluation is the most common example of an evaluation order which gives rise to non-strict semantics. Lazy evaluation is an "improvement" of call-by-name which requires that the evaluation of argument terms is shared (not repeated) in the reduction of a function application.

Non-strict semantics and lazy evaluation provide capability for more forms of recursion than those allowed by strict semantics. Two examples of such recursion are the recursive fibonacci numbers and the simple recursive fixed point:

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)

fix f = f (fix f)
```

Listing 4.7. Recursion that is invalid under strict semantics but valid under non-strict semantics.

Although the examples above can be viewed distinctly as recursion on data and recursion on functions, they both use the same underlying mechanism of non-strict semantics.

The relevance of non-strict semantics to graph algorithms comes in two forms. First, it introduces powerful forms of recursion to use in graph algorithms and cyclic data structures. Second, and more significant to this thesis's stated problem, non-strict semantics as a default can induce large space and time overheads. To mitigate these costs, Haskell compilers typically use a form of *strictness analysis* pioneered by Wadler and Hughes [62], but how much can that help algorithms on large graphs? In comparison to traditional graph drawing algorithms, at what graph size does non-strictness result in an impractical running time? Furthermore, can this be easily improved with manual strictness annotations? These questions are addressed in Chapter 5.

4.1.6. Immutable State

Along with non-strict semantics, immutable state may have the most significant effect on algorithm design. Haskell is a *pure* functional programming language, meaning named variables in Haskell are *immutable objects*, subject to *single assignment*. This calls for drastic changes in concepts from imperative languages for the sake of both efficiency and code readability.

Perhaps the most visible effect is the loss of looping constructs dependent on variable assignment. Haskell has no `for` loop, the common control flow which lies at the core of most graph algorithms. The nearest thing in name is `forM`, though it requires underlying monadic structure in the result of the loop. If reassignment is used in `forM`'s action, then the entire function-call chain from the use of `forM` up becomes locked into the `IO Monad`. Such practice is considered non-functional; hence, alternative looping constructs are encouraged.

How then might graph algorithms heavy in control flow be reorganized to work in an immutable language? Generally, the underlying data structures themselves are studied to abstract out common methods of traversal and transformation. Often these methods have common structure with similar methods used in other data structures, and the operations are then classified into standard type classes. The standardized operations then become the common vocabulary for data traversal and transformation.

As an example of how loop-like control flow works in Haskell, consider the example of iterating over a sequence of items. If the sequence is array based, with constant-time arbitrary access via integer indices, we could emulate a `for` loop by generating a list of integers from 0 to the length of the array minus one and then traverse that list one item at a time, extracting for each integer `i` the item in the array's `i`'th index. Such a process is pointless; if we can traverse a sequence one item at a time, as with the integer list, then we have no need for integer-based indexing for linear traversals.

To consider how this might apply to graph algorithms, we'll look further at the most common traversal and data transformation operations on lists. There are two primary ways to reduce a list down to a value, each with a strict and a non-strict implementation:

- *right fold* Apply a transformation function `f` from the right end of the list to the left: $(f x_1 (f x_2 \ldots (f x_n)))$.
- *left fold* Apply a transformation function `f` from the left end of the list to the right: $(f \ldots (f (f x_1) x_2) \ldots x_n)$.

If the reduction resembles a transformation from a `List a` to a `List b`, then two common approaches are as follows:

- *map* Apply a transformation function `f` to each item in a list, producing a new list.
- *zipWith* Apply a combining function `f` to each item `(a,b)` from the simultaneous traversal of two lists.

These operators, in addition to others such as the dual of fold, generalize to other data structures such as arrays and binary ordered trees. In fact, the shape of an algebraic data type determines universal properties from which various traversal and transformation operators can be extracted [5, 31, 44]. Designing algorithms with these constructs, as opposed to imperative mutable constructs, allows for *equational reasoning* [14], where program proofs are often short and composite.

Systematically exploring the above traversal and transformation operators for various graph types lies beyond the scope of this thesis, but immutability does have other, more relevant, impacts on graph algorithm concepts. Depth-first search and breadth-first search provide a foundation for many efficient graph algorithms [59]. Both of those traversal strategies require marking a node as visited. More generally, graph algorithms often rely on mutable updates of various data fields in the vertices and edges of graphs. In an immutable environment this entails much data copying and the threading of state through function calls. Although monads alleviate the syntactic tedium of state threading, their creation and use in complex situations is non-trivial. With this in mind, it is worth studying how graph drawing algorithms perform when implemented as-is in a pure language and how conceptual changes in the algorithms may affect their performance.

4.2. A Case for Graph Drawing in Haskell

Graph support in Haskell is present, but arguably weak. Haskell has two graph libraries. Both are based on promising results—*Data.Graph*, based on King and Launchbury's lazy linear

depth-first search [41], and *FGL*, based on Erwig's inductive definition of graphs [23]. Despite their successes, these results have incited surprisingly little subsequent work. Data.Graph contains a total of 6 graph algorithms: depth-first search, topological sort, connected and strongly connected components, reachability, and biconnected components. FGL contains a more impressive feature set—the 6 from Data.Graph plus 12 more: articulation point, breadth-first search, Dijkstra's shortest path, dominators, maximum independent set, minimum spanning tree, Edmonds-Karp max flow and two variations on it, transitive closure, voronoi diagram, a parameterized graph fold, and a monadic graph transformer.

Even with FGL's more expansive collection of algorithms, a July 2009 search through Haskell's central package repository, HackageDB, revealed only 10 out of 1415 packages use FGL. A look through their source code further revealed that few make special use of the inductive definition of graphs provided by FGL. Furthermore, most of the papers citing Erwig's [23], such as [11, 25, 24, 43] make no special use of the graphs as inductive data types nor of the simplified equational reasoning it provides. No paper that cites [23] adds new functional graph algorithms to FGL or expands it in any way.

There exist no functional implementations of algorithms for drawing general or planar graphs. FGL's Voronoi diagram may be the closest thing to it in FGL; however, the code is undocumented and the algorithm is not described in [23]. Two functional drawing algorithms exist for two special families of graphs. Gibbons [30] derives an algorithm for the "tidy drawing of unlabelled binary trees" using an elegant calculational approach to program construction. Kennedy [40] describes a functional algorithm in Standard ML for drawing labeled rose trees. These two algorithms provide motivation to increase graph algorithm research in a non-strict pure functional setting by continuing with other simple graph drawing algorithms as a foundation.

There exist no functional algorithms for measuring the aesthetics of graph drawings. Furthermore, regardless of implementation language, there appears to be no freely licensed

toolkit targeted specifically toward measuring aesthetics of graph drawings. Haskell, with its equational reasoning and highly structured mathematical type classes from user-contributed packages, provides an excellent environment in which to implement such a tool.

Consider a small sample of the multitude of graph families: bipartite, chordal, claw-free, cubic, DAG, line, planar, star, and so on. Algorithms and data structures for detection, application, representation, and drawing of these families remain wholly unexplored in a non-strict pure functional model. Considering the emphasis placed by functional programming on type representation and universal properties entailed by representation, this model appears to offer a structured approach to discovering further insights into the structure of graphs and graph algorithms.

CHAPTER 5

FORCE-DIRECTED PLACEMENT AND AESTHETIC MEASUREMENT IN HASKELL

In this chapter I describe my original contributions in this thesis. Among these contributions are a functional implementation of Fruchterman and Reingold's force-directed algorithm [26]; subjective and empirical validation that the functional implementation is reasonably correct; a functional implementation of Cormen, Leiserson, Rivest, and Stein's division-free segment intersection algorithm [12] modified to count edge crossings; a division-free algorithm for determining if two segments are approximately parallel; and an output-sensitive efficient algorithm for counting the maximum number of approximately parallel edges in a graph drawing.

5.1. FR91 in Haskell

Number crunching applications such as the spring-embedder algorithm are naturally suited for efficient computation in imperative languages with mutable variables, but using reassignment in Haskell is discouraged except as a last resort. We will take a mostly top-down look at a way to restructure a simple force-directed algorithm for non-strict pure functional programming.

Using the vocabulary introduced in Chapter 2, our graph representation will be straight-line drawings; our technique will be the FR91 algorithm (Section 3.4); and our aesthetics for consideration will be number of crossings, uniformity of edge lengths, and mean edge length.

5.1.1. Random Node Placement

The first step in FR91 assigns random positions to each vertex to transform an abstract graph into an initial drawing. Randomness initially seems dependent on some nondeterministic

external state; however, practice reveals that many applications perform adequately with a lenient form of pseudo-randomness. This means we can avoid locking the algorithm into the `IO Monad` at the first step [1].

The `Data.Vect` package provides instances of the standard type class `Random a` for vectors of dimension two, three, and four. All instances of `Random a` automatically support generation of an infinite stream of random values of type `a` with the values bound inside any given range. To place *n* vertices in pseudo-random initial positions on a 2D drawing frame, we simply need take *n* values from an infinite stream of 2D vectors bound within the width and height of the frame.

```
defaultDrawing :: (DynGraph gr, Vector v, Random v)
  => v -> gr l m -> gr v m
defaultDrawing frame g = mkGraph vs' (labEdges g) where
  n = noNodes g
  r = (zero,frame)
  vs' = zip (nodes g) $ take n $ randomRs r $ mkStdGen 1
```

Listing 5.1. Replacing a graph's vertex labels with pseudo-random position vectors. *zero* is the **0** vector with the same dimension as *v*.

Note that varying the frame size introduces nondeterminism into the generated coordinates without the need to modify the function. Alternatively, the seed given to `mkStdGen` could be modified. Note further that `defaultDrawing` is polymorphic in the vector *v*.

5.1.2. The Outer Loop

The first major obstacle appears in the second step of FR91 wherein we enter the primary outer loop for some fixed number of iterations. Our graph is immutable; hence, so are the vectors at its vertex labels. How, then, can we model the process of repeated improvement to the vertices' positions? The solution is to model the outer loop as a potentially infinite sequence of graphs. Traversing linearly through the sequence represents observing the refined layouts at each step of the algorithm's outer loop. As long as we do not hold onto any single

element of the list for too long, the garbage collector can destroy previously traversed graphs in the list. Increasing the iteration count should therefore not increase memory usage.

Since portions of the algorithm, such as the linear cooling function, depend on the current iteration count, this number must be passed into the primary layout function. To do this we create a modified version of the built-in `iterate` function.

```
draw :: Vec2 -> Gr Vec2 b -> [Gr Vec2 b]
draw frame = iterateNumbered (updateLayout frame)

iterateNumbered :: (Integral i) => (i -> a -> a) -> a -> [a]
iterateNumbered f x = x:iterateNumbered' 1 x where
  iterateNumbered' n c = v:(iterateNumbered' (n+1) v) where
    v = (f n c)
```

Listing 5.2. *draw* represents an infinite sequence of layout refinements; *iterateNumbered* is just like *iterate* but passes the current iteration count; *updateLayout* is the primary layout function for FR91, shown in Listing 5.9.

Asking the client of the library to manually discard irrelevant iterations from the infinite sequence would prove tedious. The function call `skip i` takes a potentially infinite list and returns another potentially infinite list containing the items at indices $[0, 1(i+1), 2(i+1), \ldots]$.

```
skip :: Int -> [a] -> [a]
skip _ [] = []
skip 0 xs = xs
skip n (x:xs) = x:skip' 0 xs where
  skip' _ [] = []
  skip' i (x:xs) | i == n = x:(skip' 0 xs)
  skip' i (x:xs) = skip' (i+1) xs
```

Listing 5.3. *skip n* skips every *n* items in the list *xs*.

To take every fifth iteration of FR91 from 0 to 50, i.e. layouts $[0, 5, \ldots, 50]$, do as shown in Listing 5.4:

```
take 11 $ skip 4 $ draw frame g
```

Listing 5.4. Every fifth iteration of FR91 from 0 to 50.

5.1.3. Computing Repulsive and Attractive Forces

Returning to the primary iteration loop, the next step involves iterating over every vertex pair (a, b) to compute repulsive forces. This is the most computationally expensive part of the algorithm. The obstacle here appears to be the lack of looping constructs. As before, all named values are immutable, so the original algorithm's strategy for updating each vertex's displacement vector inside of two nested `for` loops appears unusable:

```
for v in V
  displacement[v] = 0
  for u in V-{v}
    separation = v - u
    displacement[v] += normalize(separation)*f_r(length(separation))
```

Listing 5.5. Pseudocode for computing repulsive forces imperatively.

To find a functional equivalent, it helps to think about what we want to compute rather than how we want to compute it. In this case, we'll start with the imperative pseudocode and work backwards to what it is computing. The two `for` loops generate all pairs of vertices $(a, b) \in V \times V$ such that $a \neq b$. In FR91, both the attractive and repulsive forces satisfy the equality $f(a, b) = -f(b, a)$; hence, we really only need the 2-subset $\binom{V}{2}$ or more constructively the set $\{(a, b) : (a, b) \in V \times V \wedge a < b\}$. In Haskell we can generate these pairs as a list instead of a set. We can then iterate over the pairs, via a fold, computing the repulsive force and updating an appropriate persistent structure.

Observing that every edge $(a, b) \in E$ occurs in the above list of pairs, we can combine the second inner loop, for attractive forces, into the first. This removes a long sequential computation process from the pipeline for dense graphs, but introduces overhead for conditional edge checking in sparse graphs. The resulting imperative pseudocode for our current observations looks as follows:

The only state change occurs in the displacement array D. With a persistent array, this state can easily be passed along through the fold which acts as the single `for` loop. This

```
Let D = an array of displacement vectors all initialized to 0
Let S = all pairs (a,b) in V such that a < b
For each (a,b) in S
  separation = b - a
  dist = length(separation)
  unit = normalize(separation)
  repulsive = unit * f_r(dist)
  attractive = if (a,b) in E then (unit * f_a(dist)) else 0
  displacement = repulsive + attractive
  D[a] += displacement
  D[b] += -displacement
```

Listing 5.6. New imperative pseudocode for computing repulsive and attractive forces. Assumes $V = \{n : n \in \mathbb{N} \wedge n < |V|$.

looping construct is shown in Listing 5.7. Closing `accumDisplacement` over arguments for computing repulsive and attractive forces yields an accumulator function which can be used in a left fold. The function `computeDisplacement` folds a given accumulator function over all 2-subsets of vertices.

```
type VPair v = ((Node,Node),(v,v))
type VForceFun v = VPair v -> v
type DisplAccumL v = Array Int v -> VPair v -> Array Int v

computeDisplacement :: (Vector v, DotProd v, AbelianGroup v)
  => DisplAccumL v -> [(Node,v)] -> Array Int v
computeDisplacement fAccum vs = ds' where
  ds = zeroArray (length vs)
  ds' = foldl' fAccum ds (twoSubSet vs)

accumDisplacement :: (AbelianGroup v)
  => VForceFun v -> VForceFun v -> DisplAccumL v
accumDisplacement vF eF arr vp@((a,b),_p) = accum (&+) arr fs where
  repulsive = vF vp
  attractive = eF vp
  displ = repulsive &+ attractive
  fs = [(a,displ),(b,neg displ)]
```

Listing 5.7. The imperative loop from Listing 5.6 expressed in Haskell. Some syntactic overhead is introduced to allow custom force functions and to deal with the distinction between a vertex's label (type *Node*) and its position (type *v*).

5.1.4. Cooling and Bounding

The final inner loop limits the maximum displacement of each vertex within the bounds of the frame and the current cooling temperature. It then updates the vertex positions using the bounded displacement. The original Fruchterman and Reingold algorithm [26] centered the frame at the origin, but in Listing 5.8 the lower left corner of the frame is placed at the origin. This simplifies the algorithm's implementation by letting us take advantage of the algebraic structure of real vectors as groups—namely, polymorphic use of the additive identity **0**.

```
updatePosition :: (Surface v, DotProd v)
  => MaxDispl -> v -> v -> v -> v
updatePosition maxDisp frame pos displacement = pos' where
  disp' = (normalize displacement) &* (min maxDisp (len displacement))
  pos' = bound frame (pos &+ disp')

bound :: (Surface v) => v -> v -> v
bound frame pos = pointwiseAp max zero $ pointwiseAp min frame pos
```

Listing 5.8. *updatePosition* computes new vertex positions within the bounds of the cooling temperature (*maxDispl*) and the frame.

Putting the functional representations for the three inner loops together to represent an entire iteration leads to Listing 5.9. This is a pure, functional implementation of FR91. The implementation accepts custom force computing functions, an extension responsible for the majority of the lines of code.

This abstraction can then be wrapped in a function computing the original algorithm from [26]. Listing 5.10 shows the implementation along with the scalar force functions used in the original paper.

Observe that every function in this implementation is polymorphic in the vector **v**; hence, the same code will compute FR91 in any higher dimension. Package `Data.Vect` provides vector implementations for two, three, and four dimensions, meaning the algorithm works without modifications in those dimensions. To extend FR91 to yet higher dimensions, one need to simply implement appropriate type class instances for higher dimension vectors in the

```
updateLayout :: (Surface v, DotProd v)
  => ForceFun -> ForceFun -> IdealDistance -> v
     -> IterCount -> Gr v b -> Gr v b
updateLayout sVF sEF k frame i g = mkGraph vs' (labEdges g) where
  n = noNodes g
  vF = vecDispl sVF
  eF = vecDispl sEF
  fAccum = accumDisplacement vF eF
  ds = computeDisplacement fAccum (labNodes g)
  maxDispl = cool frame k i
  updatePos = updatePosition maxDispl frame
  vs' = map (\(n,p) -> (n,updatePos p (ds!n))) (labNodes g)
```

Listing 5.9. The entire inner loop of FR91, abstracted to support custom force functions, ideal vertex separation distance, and frame size.

```
drawFR91 :: (Surface v, DotProd v) => v -> Gr v b -> [Gr v b]
drawFR91 frame g = iterateNumbered layoutFun g where
  idealDistanceK = idealDistance frame (noNodes g)
  sVF = repulsiveForce' idealDistanceK
  sEF = attractiveForce' idealDistanceK
  layoutFun = updateLayout sVF sEF idealDistanceK frame

repulsiveForce' :: IdealDistance -> Distance -> ScalarDisplacement
repulsiveForce' k d = negate (k^2)/d

attractiveForce' :: IdealDistance -> Distance -> ScalarDisplacement
attractiveForce' k d = d*d/k

idealDistance :: (Surface v) => v -> Int -> IdealDistance
idealDistance v n = sqrt ((area v)/(fromIntegral n))
```

Listing 5.10. A pure, functional implementation of the original FR91 algorithm.

`Data.Vect` package. Appropriate projection functions to produce a 2D drawing would also need to be written, but the algorithm itself would remain unchanged.

5.2. Evaluating Functional FR91

With a functional implementation of FR91 in place, two questions arise:

- Does it produce similar results to a well-known imperative implementation of the same algorithm?
- What size graphs can it draw in reasonable time in comparison with a well-known imperative implementation of the same algorithm?

5.2.1. Comparison of Results

The resulting drawings of two implementations can be compared both subjectively and objectively. These correspond to observation of the drawings and measurement of aesthetics respectively. For comparison I use the imperative implementation of the Fruchterman and Reingold algorithm from the Boost Graph Library [55]. Like my implementation, Boost's supports custom force and cooling functions, but it does so using parametric polymorphism via C++ templates. It additionally supports a parameter to customize the generation of vertex pairs, allowing spatial indexing such as the grid variant presented in [26].

I will refer to my functional implementation as *HSFR* and to the Boost implementation as *BGLFR*. HSFR was modified to perform 100 iterations to match those of BGLFR. The linear cooling function of HSFR was modified to match that of BGLFR, which behaves as follows:

$$\max\left\{0, \frac{\text{final iteration \# - current iteration \#}}{10}\right\} \tag{7}$$

Two graph data sets are used for comparison. One is a set of 212 random graphs freely available from GDToolkit's test suite. The 212 random graphs are biconnected, undirected,

and planar. The number of vertices in the graphs range from 10 to 100. Following common practice for this set of graphs, I will refer to these 212 as the *Rome graphs*. The other is a small set of simple graphs that are easy to analyze by sight. I will denote these as the *Texas graphs*.

5.2.1.1. The Texas Graphs

An initial look at the Texas graphs, a small set of well-known graphs, indicates that the functional implementation may behave similarly to the original algorithm presented by Fruchterman and Reingold and to Boost's contemporary implementation of the algorithm (Figure 5.1). Several of the graphs, such as C_4, K_5, $K_{1,5}$, Q_3, and $K_{2,3}$ look nearly identical. The only apparent differences are small changes in orientation. On the other hand, others appear notably different, such as C_5, K_9, K_{10}, and $K_{3,3}$. At this stage of the analysis, we will assume these are anomalies due to different initial positions, which are chosen randomly, between the algorithms.

From these elementary graphs, it appears that both implementations produce the same number of crossings in their drawings in typical cases. Furthermore, these drawings appear similar to drawings of the same graphs displayed in [26]. Nevertheless, HSFR may be hiding bugs that could produce significantly different drawings in more complex graphs. A quantitative look at the resulting drawings would help to confirm the correctness of the functional implementation.

5.2.1.2. The Rome Graphs

Although HSFR treats graphs with parallel edges and loops as identical to their largest simple subgraphs, BGLFR does not. BGLFR exhibits anomalous behavior in the presence of loops; several vertices become stuck at identical positions in the corners of the frame. This happens because the attractive force is 0 and the repulsive force undefined on a loop

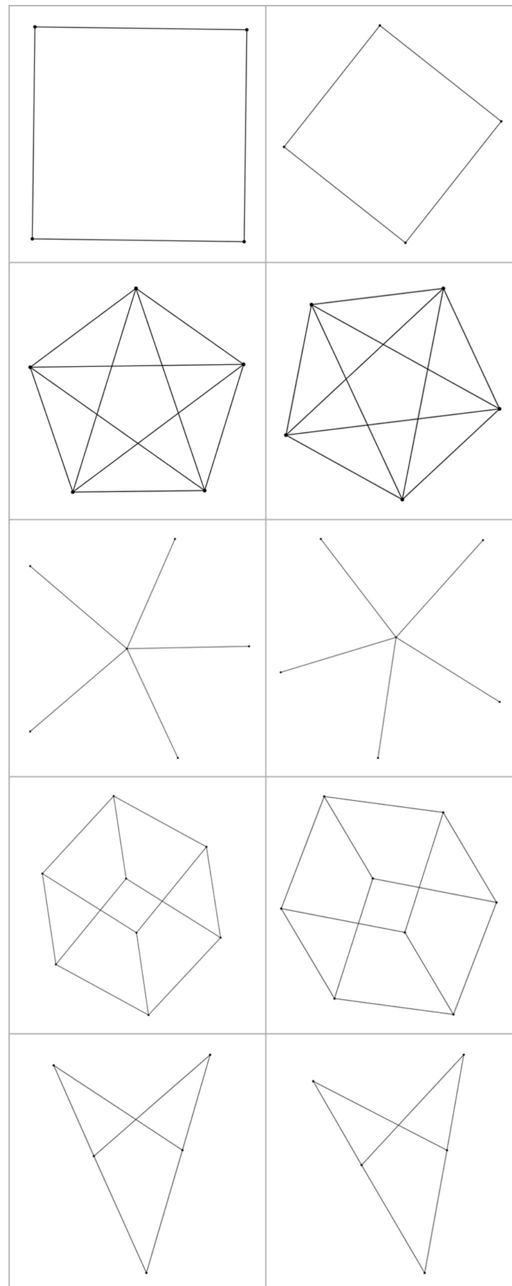

Figure 5.1. Similarities between HSFR and BGLFR.
HSFR drawings on the left; BGLFR drawings on the right. From top to bottom: C_4, K_5, $K_{1,5}$, Q_3, and $K_{2,3}$.

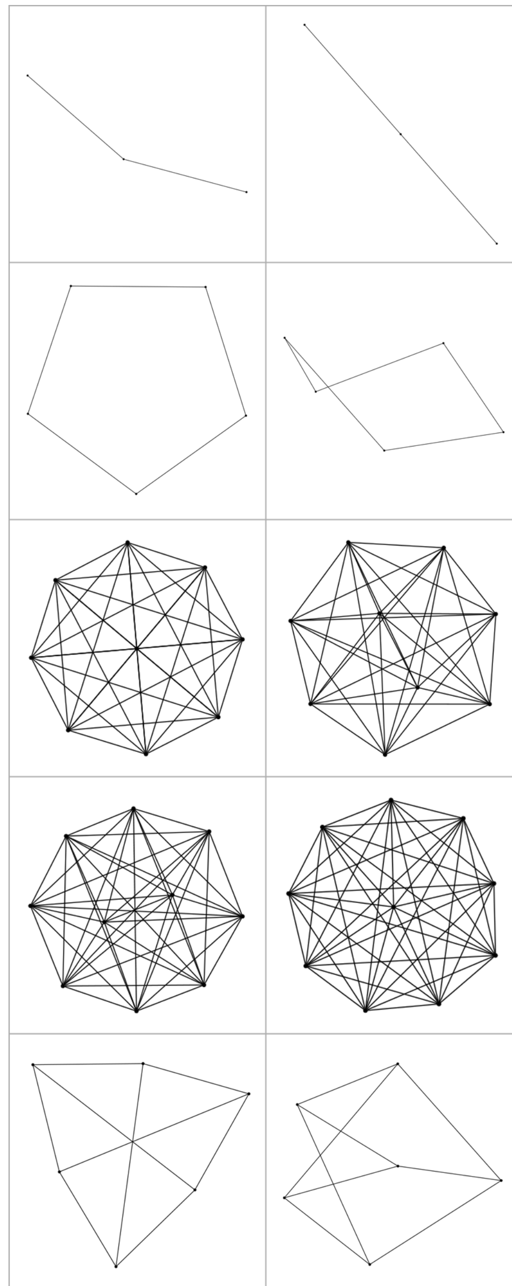

Figure 5.2. Differences between HSFR and BGLFR.
HSFR drawings on the left; BGLFR drawings on the right. From top to bottom: $K_{1,2}$, C_5, K_9, K_{10}, and $K_{3,3}$.

vertex. Since many of the Rome graphs have loops and parallel edges, the graphs were first transformed into their maximum simple subgraphs.

The tools described in Section 5.3.1 were used to collect empirical data on the resulting drawings from both implementations. Since crossings are a strongly emphasized aesthetic, we will look at them first. Crossing measurements for HSFR are shown in Table 5.1. Among the 212 graphs, drawings with 2 crossings resulted more frequently than any other number. The average was 229 crossings with a standard deviation of 198. Hence, the number of crossings varied significantly across the drawings. At least half of the drawings have 161 or more crossings. The minimum number of crossings in any drawing is 0 and the maximum is a shocking 709. Recall that all 212 of the Rome graphs in this set are planar. Looking at the number that deviated from the average by more than the standard deviation tells us which graphs were drawn unusually well for this implementation and which were drawn unusually poorly. For HSFR, 44 were drawn exceptionally poorly while 40 were drawn exceptionally well (disregarding the fact that the graphs are planar).

Mean	229.0943
σ	198.1690
Median	161
Mode	2
Extrema	(0,709)
value − mean $> \sigma$	44 times
value $<$ mean − σ	40 times

Table 5.1. Crossing measures for HSFR.

For BGLFR, drawings with 3 crossings resulted more frequently than any other number. The average was 250 crossings with a standard deviation of 222. As with HSFR, the number of crossings varied significantly across the drawings. At least half of the drawings have 176 or more crossings. The minimum number of crossings in any drawing is 0 and the maximum is an even more shocking 904. Forty-four graphs were drawn exceptionally poorly while 40 were drawn exceptionally well (again disregarding the fact that the graphs are planar).

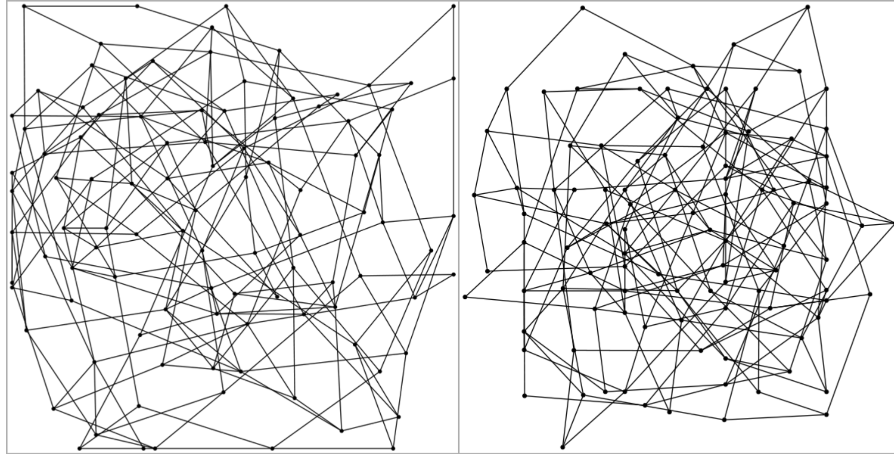

Figure 5.3. Worst crossings for both implementations.
The same graph results in the maximum crossings in any drawing for BGLFR (914, right) and HSFR (709, left).

Mean	249.5425
σ	222.1664
Median	176
Mode	3
Extrema	(0,904)
value $-$ mean $> \sigma$	35 times
value $<$ mean $- \sigma$	39 times

Table 5.2. Crossing measures for BGLFR.

This data begins to confirm the intuition that spring-embedder algorithms frequently sacrifice crossing optimization for the sake of uniform edge lengths. In its current form, however, the data does not explicitly indicate that the two implementations behave similarly. To get more insight into similarity, we can look at statistical measures taken over the pairwise difference of crossings for drawings of identical graphs. Table 5.3 displays these measures.

BGLFR and HSFR produce the same number of crossings in drawings of the same graphs 7 times. BGLFR produces more crossings than HSFR 127 times, while the converse happens 78 times. The average difference in the number of crossings between two drawings of the same graph is roughly 20 with a standard deviation of 49. At least half of the drawings differ by no more than 5 crossings, while the most common difference between crossings in pairwise

drawings is 2. The extrema represent the extreme values of the differences in both directions; hence, in the worst pairwise comparison HSFR produced a drawing with 93 more crossings than BGLFR's drawing of the same graph. Likewise, BGLFR produced a drawing with 239 more crossings than the corresponding drawing by HSFR.

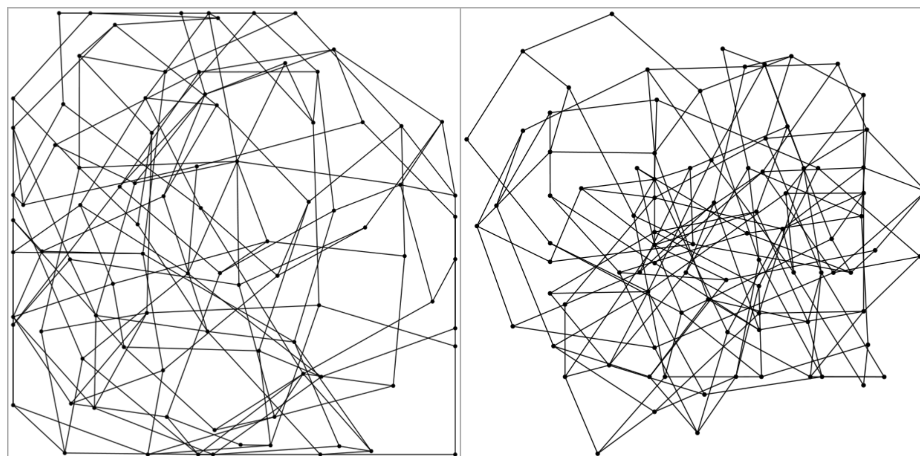

Figure 5.4. BGLFR's largest excess of crossings.
Not easily observed by sight, HSFR (left) produced 476 crossings while BGLFR (right) produced 715, a difference of 239.

With a casual glance, we may be tempted to conclude that HSFR somehow generally produces drawings with fewer crossings than BGLFR; howevever, such a conclusion would be erroneous. The average difference is less than half of the standard deviation and a mere 49 out of 212 pairs of drawings lie outside of this interval. In conclusion, both implementations appear to behave similarly with respect to the number of crossings in resultant drawings. Unfortunately, this behavior is rather poor. Common folklore maintains that simple force-directed algorithms are mainly suitable for small graphs with 50 or fewer vertices. These measurements confirm that folklore. Large graphs, even planar ones, practically always suffer poor vertex placement under a force-directed algorithm.

Since the most prominent aesthetic in FR91, and force-directed algorithms in general, is uniform edge length, we will next observe measurements for it. We will look at measurements performed on edge lengths normalized so that the longest length is 1.0.

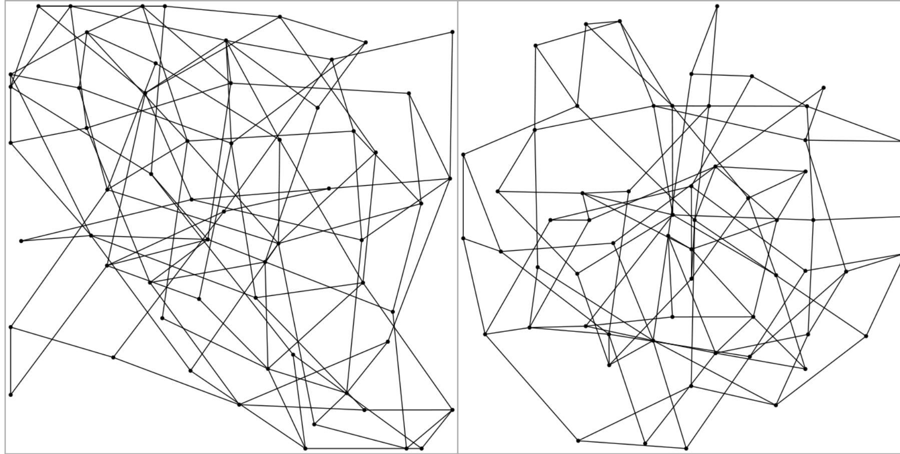

Figure 5.5. HSFR's largest excess of crossings.
Not easily observed by sight, HSFR (left) produced 294 crossings while BGLFR (right) produced 201, a difference of 93.

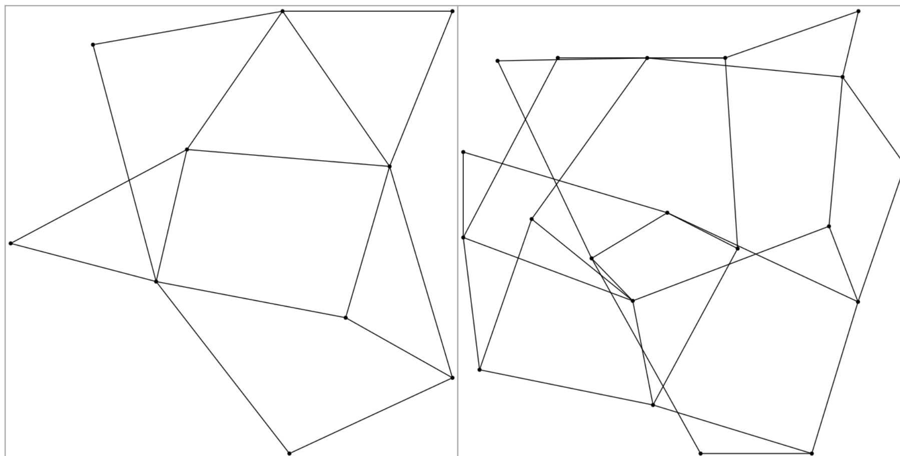

Figure 5.6. Most and least uniform edges for a BGLFR drawing.
BGLFR's drawing with most uniform edge lengths (left) and least uniform edge lengths (right).

$A = B$	7 times
$A < B$	127 times
$A > B$	78 times
Mean	20.4481
σ	48.5165
Median	5
Mode	2
Extrema of $(B - A)$	(-93,239)
$(B - A) > \sigma$	43 times
$(B - A) < -\sigma$	6 times

Table 5.3. Differences in crossing measures.

Difference in crossing measures between BGLFR and HSFR. A denotes the number of crossings in a drawing for HSFR and B denotes the number in a drawing of the same graph for BGLFR.

Taking the standard deviation of edge lengths in a given drawing provides a measure of their uniformity. A standard deviation of 0 would indicate perfectly uniform edge lengths. Taking statistical measures of this standard deviation over all drawings represents a measure in the variation of uniformity among all 212 drawings. For both HSFR and BGLFR, no perfectly uniform drawing was produced; the closest was HSFR with a drawing having edge length uniformity 0.12. Neither implementation draws excessively non-uniform edge lengths and both deviate little in their overall uniformity. These conclusions arise from Table 5.4 and Table 5.5.

To summarize, both HSFR and BGLFR behave similarly with respect to optimizing uniform edge lengths. The extent of uniformity in edge lengths provides additional evidence for the folklore intuition that modeling springs with uniform resting lengths leads to highly uniform edge lengths at the expense of increasing crossings.

Before concluding that HSFR correctly implements the Fruchterman and Reingold algorithm as it is understood, we should analyze whether drawings produced by the two implementations have roughly the same edge lengths as each other. This time we will look at

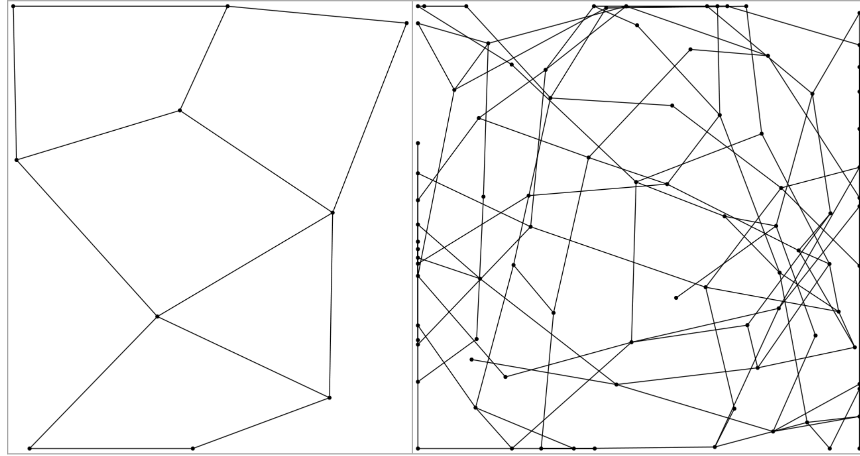

Figure 5.7. Most and least uniform edges for a HSFR drawing.
HSFR's drawing with most uniform edge lengths (left) and least uniform edge lengths (right).

Measure	HSFR	BGLFR
Mean	0.1931	0.1840
σ	0.0179	0.0139
Median	0.1937	0.1856
Mode	0.1932	0.1885
Extrema	(0.1238,0.2318)	(0.1275,0.2202)
value − mean $> \sigma$	24 times	88 times
value $<$ mean $- \sigma$	24 times	18 times

Table 5.4. Measures of normalized edge lengths.

Measures over normalized edge length σ for HSFR and BGLFR.

measurements based on non-normalized edge lengths in a 500×500 unit frame. If two separate drawings of the same graph have roughly the same mean edge length, then the drawings probably look similar.

As Table 5.6 shows, the mean edge length varies little between two drawings of the same graph. To summarize, both implementations produce equivalently poor crossing aesthetics while behaving measurably similarly in edge length uniformity and mean edge length. Considering these measurements, we can safely conclude that the non-strict, pure functional implementation of the algorithm is reasonably correct.

$A = B$ within $\epsilon = 10^{-6}$	0 times
$A < B$ i.e. $(A - B) < -10^{-6}$	61 times
$A > B$ i.e. $(A - B) > 10^{-6}$	151 times
Mean	-0.0091
σ	0.0194
Median	-0.00905
Mode	-0.0165
Extrema of $(B - A)$	(-0.0556,0.0621)
$(B - A) > \sigma$	12 times
$(B - A) < -\sigma$	57 times

Table 5.5. Difference in normalized edge length standard deviations.

Difference in normalized edge length standard deviation between BGLFR and HSFR. A denotes edge length σ in a drawing for HSFR and B denotes the same in a drawing of the same graph for BGLFR.

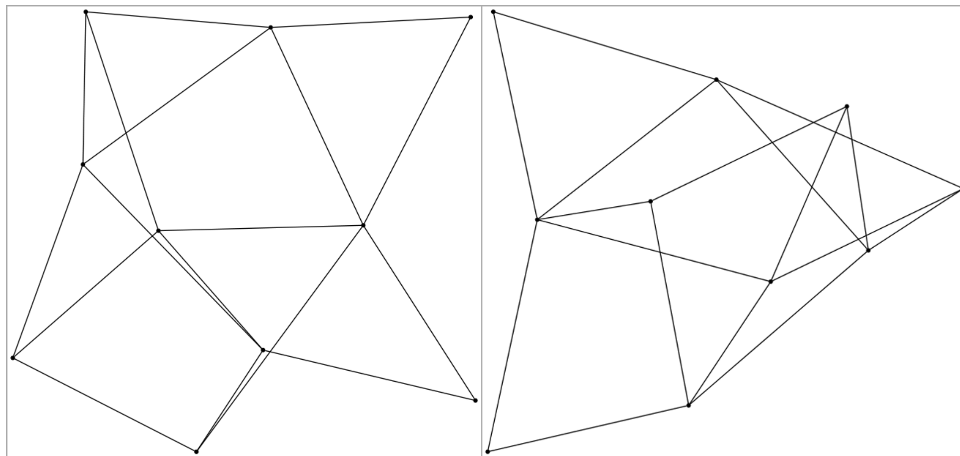

Figure 5.8. Typical outcome of HSFR and BGLFR implementations.
At least half of the drawings produced by HSFR (left) and BGLFR (right) differ by no more than 5 crossings.

5.2.2. Comparison of Running Times

As one might expect, the initial functional implementation described in this thesis is not at all competitive with BGLFR in performance on large graphs. Table 5.7 shows measures of user time on the Rome graphs classified by vertex count.

Although the measured user times do not bode well in comparison to BGLFR, they provide a direction for context in which the current implementation is usable. Specifically, for planar

$A = B$ within $\epsilon = 10^{-6}$	0 times
$A < B$ i.e. $(A - B) < -10^{-6}$	9 times
$A > B$ i.e. $(A - B) > 10^{-6}$	203 times
Mean	-18.8040
σ	8.9320
Median	-20.37515
Mode	-17.5066
Extrema of $(B - A)$	(-33.3659,15.2261)
$(B - A) > \sigma$	1 times
$(B - A) < -\sigma$	178 times

Table 5.6. Difference in mean edge length.

Difference in mean edge length pairwise between BGLFR and HSFR. Frame width is 500×500 units. A denotes mean edge length in a drawing for HSFR and B denotes the same in a drawing of the same graph for BGLFR.

graphs of up to 40 vertices the running time of HSFR is suitable for offline testing of graph algorithms. Note that for planar graphs, $m \leq 3n - 6$. For large graphs the implementation becomes unsuitable for anything but offline drawing on small test sets or individual graphs. The result on large graphs is mainly relevant only to testing, since simple versions of force-directed algorithms provide little practical use in drawing large graphs.

I am not ambitious enough to tackle in this thesis the question of whether an equivalently fast Haskell implementation can be implemented, but I will briefly mention the direction that such a project would take. The primary cause of slow running times in Haskell programs are often space and time leaks due to non-strict evaluation. Profiling the implementation with the GHC Haskell compiler would reveal portions of the program causing excessive memory use. These troublesome areas could be analyzed and possibly annotated with strictness flags, allowing the compiler to perform certain optimizations to make the generated code more like its C counterpart. If this approach still did not produce competitive performance, the implementation could be overhauled to use unboxed, mutable arrays for the troublesome areas of the algorithm.

$\|V\|$	N	HSFR Mean	BGLFR Mean	HSFR σ	BGLFR σ	HSFR Median	BGLFR Median
10	20	0.05	0.00	0.01	0.00	0.05	0
20	20	0.23	0.00	0.01	0.00	0.23	0
30	20	0.51	0.01	0.01	0.00	0.52	0.01
40	20	0.90	0.02	0.03	0.00	0.9	0.02
50	32	1.37	0.03	0.13	0.00	1.32	0.03
60	20	2.03	0.05	0.04	0.01	2.04	0.045
70	20	2.67	0.06	0.04	0.00	2.665	0.06
80	20	3.29	0.07	0.07	0.00	3.28	0.07
90	20	4.39	0.09	0.11	0.01	4.385	0.09
100	20	5.22	0.10	0.05	0.01	5.205	0.1

Table 5.7. User time on Rome graphs.

User time on Rome graphs. N is the sample size.

5.3. A Freely Licensed Toolkit for Aesthetic Measurement of Graph Drawings

Experimental studies for graph drawing algorithms have begun to gain popularity [9, 61]; however, no freely licensed toolkit—independent from graph drawing frameworks—exists to provide a foundation for such studies.

In this section I describe a few functional implementations of aesthetic measuring algorithms. The algorithms come from a project intending to establish a common tool with which to do experimental graph drawing studies.

5.3.1. Simple Aesthetics

Fundamental operations in experimental studies include statistical measures such as mean, median, standard deviation, histogram, and so on. Although these can have rather straightforward implementations in functional languages (Listing 5.11), as shown in [46] efficiency issues can easily arise from the simplest of the computations, mean.

```
mean :: [Double] -> Double
mean xs = sum xs / length xs
```

Listing 5.11. An implementation of mean with an inefficient space leak. An efficient alternative is given in [46].

Stewart, a co-author of [46], has implemented a freely licensed high-performance statistics package in Haskell, available on the web. As described by Stewart [56], the package is "A collection of commonly used statistical functions designed to fuse under stream fusion, with attention paid to the generated assembly." This library can therefore provide the foundational statistical operations needed by a graph drawing experimental study toolkit.

Given a graph drawing, which can be parsed from some text file, most statistical measures can be prototyped quite quickly in a straightforward manner. First, the abstract edges must be transformed into line segments using vertex positions. To do this efficiently we need constant-time access to arbitrary vertex positions; therefore, the default list data structure used by functional graph algorithms will not do. The vertices can be put into an array, but the vertex labels must first be ordered contiguously. This relabeling can be done efficiently in a functional language with the use of an integer-based map rather than lists.

```
type Line = (Vec2,Vec2)

toSegments :: Gr Vec2 b -> [Line]
toSegments g = es where
  g' = relabelContiguous g
  ns = array (1,noNodes g') (labNodes g') :: Array Int Vec2
  es = map (\(a,b) -> ((ns!a),(ns!b)))
        (filter (\(a,b) -> a /= b) $ edges g')

relabelContiguous :: Gr a b -> Gr a b
relabelContiguous g = mkGraph ns' es' where
  im = IM.fromList $ zipWith (\(n,l) m -> (n,(m,l))) (labNodes g) [1..]
  es' = map (\(a,b,l) ->
         ((fst $ im IM.! a),(fst $ im IM.! b),l)) (labEdges g)
  ns' = IM.elems im
```

Listing 5.12. Transforming abstract edges into line segments. The algorithm depends on vertices having contiguous identifiers.

Computing edge lengths relies on the distance function from the `Data.Vect` package. For more reliable measures across drawings it may help to normalize the edge lengths so that the longest edge has length 1.

```
edgeLengths :: [Line] -> [Double]
edgeLengths = map (uncurry distance)

normalEdgeLengths :: [Line] -> [Double]
normalEdgeLengths = normalLengths . edgeLengths

normalLengths :: (Fractional a, Ord a) => [a] -> [a]
normalLengths xs = map (/max) xs where
  max = maximum xs
```

Listing 5.13. Simple edge length computation.

The `Control.Arrow` library, intended as an alternative to monads, provides convenient syntax for applying multiple functions to one value, used in Listing 5.14 to compute the extrema of edge lengths with little syntactic complexity.

```
edgeLengthExtrema :: [Line] -> (Double,Double)
edgeLengthExtrema = (minimum &&& maximum) . edgeLengths
```

Listing 5.14. `(&&&)` from *Control.Arrow* provides convenient syntax for applying tuples of functions to a single value.

This short list of functions combined with basic statistical operators provides all of the measures used in Section 5.2.1.2. The implementation currently supports loading of graph drawings from the Rome graph format or from a subset of the Graphviz format. It can provide a long listing which includes actual and normalized edge lengths for a single drawing (Listing 5.15) or it can compare two drawings (Listing 5.16). It can also create a new drawing which contains the aesthetic measurements in the image (Figure 5.9).

5.3.2. Testing for Edge Crossing

The cross product $a \times b$ of two vectors a and b represents a vector perpendicular to the plane that contains a and b. It is defined exclusively in 3 and 7 dimensions; however, it can be useful to extend the definition to also work in 2 dimensions.

As described in [12], cross product can also be interpreted as the signed area of the parallelogram lying in the same plane as a and b and having a and b as sides. Under this

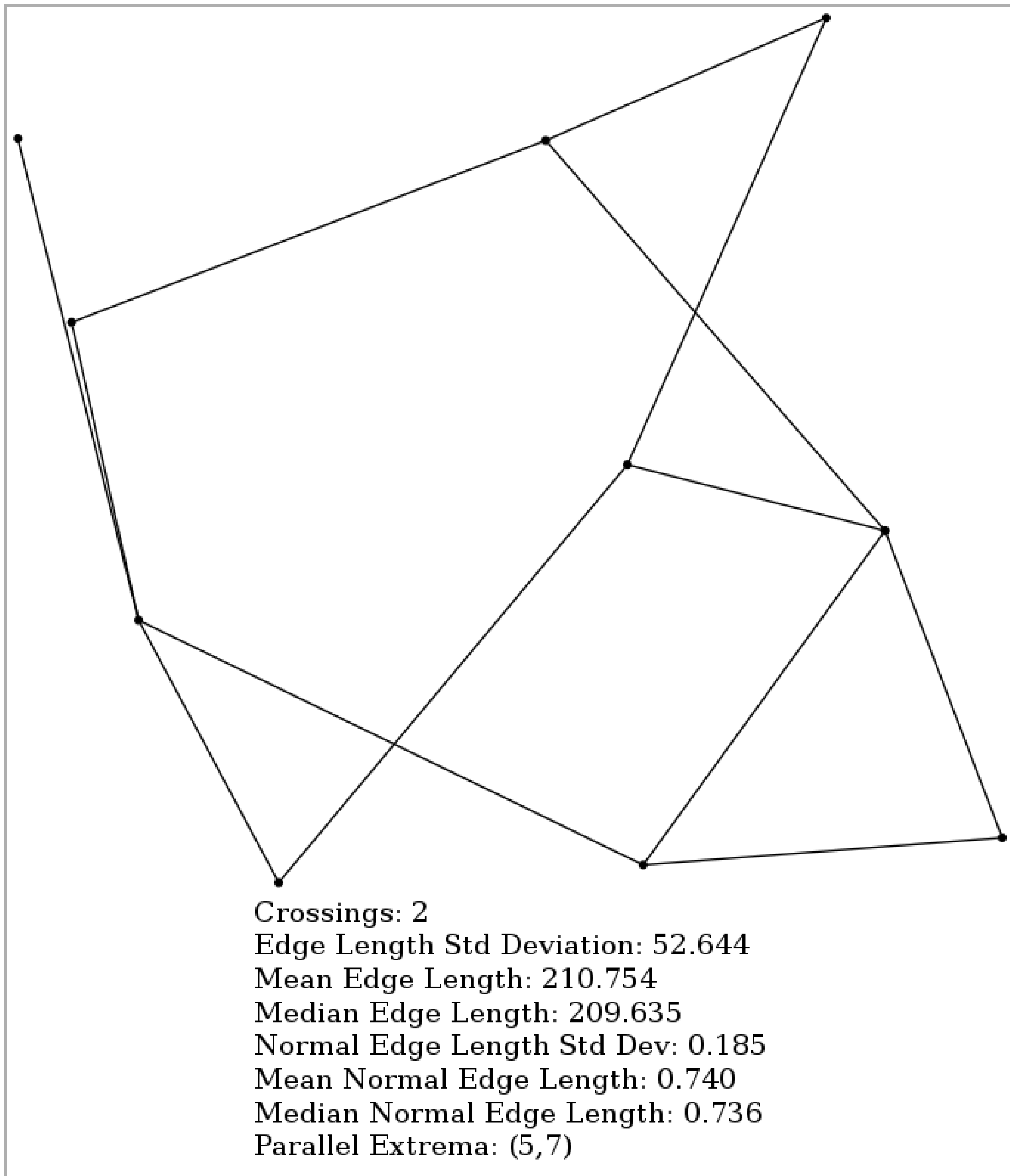

Figure 5.9. A drawing with aesthetic measures included in the image.

```
Crossings: 3
Edge Lengths: [171.453,319.148,171.421,171.823,
  318.788,171.774,171.393,318.718,171.568]
Edge Length Std Deviation: 69.444
Mean Edge Length: 220.676
Median Edge Length: 171.774
Normal Edge Lengths: [0.537,1.000,0.537,0.538,
  .999,0.538,0.537,0.999,0.538]
Normal Edge Length Std Dev: 0.218
Mean Normal Edge Length: 0.691
Median Normal Edge Length: 0.538
Parallel Extrema: (4,5)
```

Listing 5.15. A listing of some aesthetic measures for a graph drawing read from a text file.

```
Graph/Measure:      k4.fr91.simple      k4.bgl-fr.simple    Diff:
Crossings:          1                   1                   0
EdgeLenStdDev:      41.6496             41.6539             0.0043
EdgeLenMean:        242.7516            242.7763            0.0247
EdgeLenMedian:      213.3009            213.3227            0.0218
EdgeNormStdDev:     0.1381              0.1380              -0.0000
EdgeNormMean:       0.8046              0.8044              -0.0002
EdgeNormMedian:     0.7070              0.7069              -0.0002
min/max parallel:   1/2                 1/2                 0/0
```

Listing 5.16. A pairwise comparison of aesthetic measures on two separate drawings of the same graph. The second column shows the functional implementation from this thesis, the third column the Boost implementation, and the fourth column the difference between the two.

interpretation, a and b are two-dimensional vectors and the cross product $a \times b$ is a scalar instead of a vector.

The *determinant* function in algebra has an equivalent geometric interpretation for 2×2 matrices. This gives rise to an equation for computing a 2D cross product:

$$\begin{bmatrix} x \\ y \end{bmatrix} \times \begin{bmatrix} u \\ v \end{bmatrix} = det \begin{bmatrix} x & y \\ u & v \end{bmatrix} = xv - uy \tag{8}$$

Cormen, Leiserson, Rivest, and Stein [12] then use this two-dimensional cross product to derive a simple line segment intersect test which avoids division and is therefore less susceptible to floating point error than a traditional intersection test.

In functional programming, community practice encourages simplifying function definitions as much as practically possible. The cited segment testing approach allows one to avoid checking for division by 0 when either of the segments is vertical, allowing shorter code which does not need to test edge cases.

Their algorithm considers two segments to intersect if the endpoints touch. When counting edge crossings in graphs, adjacent edges are allowed to touch at their common vertex. Fortunately, the algorithm is easily modified to exclude endpoints from the intersection test:

```
segmentsCrossing :: (Vec2,Vec2) -> (Vec2,Vec2) -> Bool
segmentsCrossing (a,b) (c,d) = r where
  d1 = direction c d a
  d2 = direction c d b
  d3 = direction a b c
  d4 = direction a b d
  r  = (((d1 > 0 && d2 < 0) || (d1 < 0 && d2 > 0))
        && ((d3 > 0 && d4 < 0) || (d3 < 0 && d4 > 0)))
    || (d1 == 0 && onSegment c d a)
    || (d2 == 0 && onSegment c d b)
    || (d3 == 0 && onSegment a b c)
    || (d4 == 0 && onSegment a b d)
  direction i j k = det ((k &- i) , (j &- i))
  onSegment i j k =
       min (_1 i) (_1 j) < (_1 k)
    && max (_1 i) (_1 j) > (_1 k)
    && min (_2 i) (_2 j) < (_2 k)
    && max (_2 i) (_2 j) > (_2 k)
```

5.3.3. A Simple Algorithm for Testing Approximate Geometric Parallelism

In this section I introduce a new, parametric aesthetic and give an algorithm for computing it.

Using the cross product as defined above and in [12], testing for exact parallelism between two segments is trivial: Two segments are parallel if and only if their cross product is equal to 0. This test is largely useless though; in a drawing several edges might be almost parallel but not exactly. Expanding on the observations from [12], we can use the properties of cross product to create a division-free algorithm for testing whether any two line segments are almost parallel for any definition of almost.

5.3.3.1. Approximate Geometric Parallelism

Say that two line segments are *approximately parallel* if, when joined at one of their endpoints to form a chain with one bend, the smaller angle of the bend is less than or equal to θ for some specified θ in radians, $0 \leq \theta \leq \frac{\pi}{2}$. We call θ a *bound* on the approximated parallelism and the problem of testing whether two segments are approximately parallel *approximate geometric parallelism*.

Note that the above definition implies carefully choosing which endpoints to connect so that the bend has an angle in $[0, \frac{\pi}{2}]$. This is only for simplicity in the definition. The algorithm which follows is indifferent to which endpoints are "joined" (i.e. moved to the origin).

Let $a = (\begin{bmatrix} s_x \\ s_y \end{bmatrix}, \begin{bmatrix} t_x \\ t_y \end{bmatrix})$ and $b = (\begin{bmatrix} u_x \\ u_y \end{bmatrix}, \begin{bmatrix} v_x \\ v_y \end{bmatrix})$ be two line segments in the plane.

Move both segments to the origin so that they are represented by only two vectors:

$$\mathbf{c} = \begin{bmatrix} c_x \\ c_y \end{bmatrix} = \begin{bmatrix} s_x \\ s_y \end{bmatrix} - \begin{bmatrix} t_x \\ t_y \end{bmatrix} \text{ and } \mathbf{d} = \begin{bmatrix} d_x \\ d_y \end{bmatrix} = \begin{bmatrix} u_x \\ u_y \end{bmatrix} - \begin{bmatrix} v_x \\ v_y \end{bmatrix} \tag{9}$$

The two segments are exactly parallel if $\mathbf{c} \times \mathbf{d} = 0$.

Observe that the cross product $\mathbf{c} \times \mathbf{d}$ is proportional to the sine of the smaller angle between $\mathbf{c}$ and $\mathbf{d}$:

$$\mathbf{c} \times \mathbf{d} = |\mathbf{c}||\mathbf{d}| \sin \theta \text{ where } 0 \leq \theta \leq \pi \tag{10}$$

Substituting the equivalent definition for determinant and eliminating its unnecessary sign, we now have a simple test for whether **c** and **d** are approximately parallel within θ radians:

$$|c_x d_y - c_y d_x| \leq |\mathbf{c}||\mathbf{d}| \sin\theta \tag{11}$$

The code in Haskell:

```
parallelBy :: Double -> Line -> Line -> Bool
parallelBy theta (s,t) (u,v) =
  abs (det (a,b)) <= len a * len b * sin theta where
    a = (t &- s)
    b = (v &- u)
```

Like the segment intersect algorithm, this algorithm does not use division—nor does it explicitly use the slope of the lines or the angle between them. It does, however, use square root, square, and sine, so it may be susceptible to floating point error in some situations.

5.3.4. An Efficient Output-Sensitive Algorithm Counting Maximum Approximately Parallel Edges

Consider some edge e in a graph. For a given θ, there is some number k of approximately parallel edges to e. These k edges may not be pairwise approximately parallel to each other, but they do represent a set of edges approximately parallel to some edge in the graph. Finding a maximum k among all edges $e \in E$ with a naive algorithm takes time $O(m^2)$ which can be as bad as $O(n^4)$. By computing an edge's deviation from some arbitrary fixed axis, say the x-axis,and storing it in an appropriate efficient data structure, we can design an output-sensitive efficient algorithm for finding a maximum k which takes time $O(mklog(m/k))$. For large k this algorithm is less efficient than the naive algorithm, but for small k it is much quicker. For connected graphs without overlapping vertices, k is limited by n and m.

The algorithm relies on an interval map data structure supporting $O(logn)$ insertion and $O(klog(n/k))$ search returning all intervals that contain a given point. Such a data structure is available in Haskell thanks to Hinze and Paterson [34].

We must make a slight modification to Paterson's implementation to arrive at the parallel counting algorithm. Currently, insertions of intervals $[a, b]$ with $a > b$ into the interval map produce undefined behavior. Modify the definition of interval map as follows:

- Allow specification of min and max interval bounds at creation time.
- When storing the elements into the underlying data structure, store with each element a unique integer id which is used for efficient retrieval from the underlying data structure.
- If an element is inserted with an interval that exceeds the declared bounds of the interval map, split the interval into two intervals such that the excess of the interval "wraps around" to the other side of the bounds, similar to modular arithmetic.
- In one of these intervals store the actual element together with the unique id and in the other store only the unique integer id of the element as a separate type. This can be done with the `Either` data type in Haskell. Call the `(Int,element)` entry *right* and the `Int` entry *left*.
- Modify the search operation to also "wrap around" when necessary, so that whenever a *left* entry appears in the results it is replaced by the corresponding *right* entry or merely discarded if the *right* entry was already retrieved.

Declaring the bounds at the beginning is a constant-time operation. Furthermore, inserting two intervals instead one during an element insertion also produces constant time overhead as does potentially retrieving a second interval upon appearance of a *left* element. The modified interval map with a circular fixed interval bound therefore has the same asymptotic running time as the original interval map.

Choosing the x-axis as the base for angle measurements allows us to once again use the observations of the cross product to easily compute the necessary angle. The cross product is proportional to the sine of a segment's angle from the x-axis, which is the sine of the unknown angle we must compute.

The algorithm to compute the maximum number of (not necessarily pairwise) approximately parallel edges proceeds as follows:

- Given an angle, θ, you want to find the maximum number of edges parallel within a tolerance of θ radians.
- For each edge, store that edge's angle of deviation from the x-axis, d, computed with cross product, in a modified interval map using the interval $[d - \theta, d + \theta]$.
- If the interval exceeds $\frac{\pi}{2}$ or $-\frac{\pi}{2}$ then split the interval into two intervals so that the previously excess interval becomes a new interval on the opposite end of the y-axis.
- For each edge's deviation d, get all intervals in the interval map that contain d.
- Take the maximum of the lengths of the results.

Since the algorithm uses the modified interval map only within the capacities specified above, its running time is bounded by that of the modified interval map.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

In this thesis I provided motivation for implementation of graph drawing algorithms and aesthetic measuring tools in the non-strict pure functional programming language Haskell. I described a straightforward implementation of the Fruchterman and Reingold force-directed algorithm and compared it—both subjectively and objectively—to a well-known implementation in C++. The functional implementation supports custom attractive and repulsive forces, cooling functions, frame sizes, and expected iteration counts. Furthermore, the implementation is polymorphic in its vector type, allowing the same implementation to work in arbitrary higher Euclidean dimensions. I concluded that the implementation appropriately reflects the abstract algorithm. I also asserted that future refinements on strictness could provide better performance, making room for modern improvements to the basic force-directed approach. I motivated the need for a freely licensed toolkit—independent from any graph drawing algorithm framework—to measure aesthetics of graph drawings regardless of the implementation that generated the drawing. I then described parts of an implementation of such a toolkit, implemented in Haskell. I finally described an unusual aesthetic, the number of parallel edges in a drawing, and outlined an output-sensitive efficient algorithm for computing this measure.

6.1. Future Work

Unfinished work which did not make it into this thesis includes refinements of the aesthetic toolkit to allow large-scale experimental analysis of multiple graph drawing algorithms. Steps toward this goal include the following: simplifying and automating the interface; adding more aesthetic measuring algorithms; supporting more text-based graph drawing formats;

accumulating more sample graphs, classified via diverse properties; and measuring aesthetics at each iteration of an algorithm.

With measurements taken at each iteration of an algorithm, we can empirically compare the expected running time of force-directed algorithms without relying on absolute time or machine specifications. Such measurements would also allow one to run a force-directed algorithm with the following goal: Iterate until this conjunction of criteria is met or until an unreasonable number of iterations have passed. This concept of automated aesthetic measurements also leads to the idea of having an algorithm automatically choose an approximately optimal 2D projection from higher-dimension drawings of graphs. One could specify that the projection must meet certain criteria, be it the least crossings of the considered projections or the most uniform edge length.

It goes nearly without saying that more graph drawing algorithms could be implemented in non-strict pure functional programming languages. Dozens are available to choose from, and it would be interesting to see a formal study of the trade-offs among non-strictness, immutability, and strong typing, since Haskell compilers, and parts of the language itself, typically support various perversions to each of these dogmas. Furthermore, I suspect that recent insights into the well-known but complex relationship between depth-first search and planarity [32, 16] could benefit from a strong typing perspective with recursive, higher-order types.

Finally, I believe much potential exists for specializing inductive types for various families of graphs. Such specializations could reveal unexplored *universal properties* for representations of certain types of graphs and could lead to improvements in the efficiency or semantics of inductive graph algorithms. This rises from an unconfirmed suspicion that the inductive definition presented by Erwig in [23], while clever, may not be the most direct one possible for many situations. An increase in the number of graph algorithms in Haskell, including graph drawing algorithms, would provide direction on key problems in current implementations.

REFERENCES

[1] Heinrich Apfelmus, *[haskell-cafe] re: Generate 50 random coordinates*, Haskell-cafe mailing list, December 2006, `http://www.haskell.org/pipermail/haskell-cafe/2006-December/020005.html`, Accessed August 2009.

[2] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis, *Algorithms for drawing graphs: an annotated bibliography*, Comput. Geom. 4 (1994), 235–282.

[3] Bernd Becker and Günter Hotz, *On the optimal layout of planar graphs with fixed boundary*, SIAM J. Comput. 16 (1987), no. 5, 946–972.

[4] François Bertault, *A force-directed algorithm that preserves edge-crossing properties*, Inf. Process. Lett. 74 (2000), no. 1-2, 7–13.

[5] Richard S. Bird, *Algebraic identities for program calculation*, Comput. J. 32 (1989), no. 2, 122–126.

[6] George D. Birkhoff, *Aesthetic measure*, Cambridge, Harvard University Press, 1933.

[7] Kellogg S. Booth and George S. Lueker, *Testing for the consecutive ones property, interval graphs, and graph planarity using pq-tree algorithms*, J. Comput. Syst. Sci. 13 (1976), no. 3, 335–379.

[8] John M. Boyer and Wendy J. Myrvold, *On the cutting edge: Simplified o(n) planarity by edge addition*, J. Graph Algorithms Appl. 8 (2004), no. 2, 241–273.

[9] Franz J. Brandenburg, Michael Himsolt, and Christoph Rohrer, *An experimental comparison of force-directed and randomized graph drawing algorithms*, Springer-Verlag, 1996, pp. 76–87.

[10] Franz-Josef Brandenburg, David Eppstein, Michael T. Goodrich, Stephen G. Kobourov, Giuseppe Liotta, and Petra Mutzel, *Selected open problems in graph drawing*, Proc. 11th Int. Symp. Graph Drawing (GD 2003), Lecture Notes in Computer Science, no. 2912, Springer-Verlag, September 2003, pp. 515–539.

[11] Sylvain Conchon, Jean-Christophe Filliâtre, and Julien Signoles, *Designing a generic graph library using ml functors*, Trends in Functional Programming, 2008, pp. 124–140.

[12] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein, *Introduction to algorithms, second edition*, The MIT Press and McGraw-Hill Book Company, 2001.

[13] Weiwei Cui, *A survey on graph visualization*, Phd qualifying exam (pqe) report, Computer Science Department, Hong Kong University of Science and Technology, 2007.

[14] Nils Anders Danielsson, John Hughes, Patrik Jansson, and Jeremy Gibbons, *Fast and loose reasoning is morally correct*, POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (New York, NY, USA), ACM, 2006, pp. 206–217.

[15] Ron Davidson and David Harel, *Drawing graphs nicely using simulated annealing*, ACM Trans. Graph. 15 (1996), no. 4, 301–331.

[16] Hubert de Fraysseix, *Trémaux trees and planarity*, Electronic Notes in Discrete Mathematics 31 (2008), 169–180.

[17] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis, *Graph drawing*, Prentice Hall, Upper Saddle River, NJ, 1999.

[18] Tim Dwyer, Kim Marriott, and Michael Wybrow, *Integrating edge routing into force-directed layout*, Graph Drawing (Michael Kaufmann and Dorothea Wagner, eds.), Lecture Notes in Computer Science, vol. 4372, Springer, 2006, pp. 8–19.

[19] ______, *Topology preserving constrained graph layout*, Graph Drawing (Ioannis G. Tollis and Maurizio Patrignani, eds.), Lecture Notes in Computer Science, vol. 5417, Springer,

2008, pp. 230–241.

[20] Peter Eades, *A heuristic for graph drawing*, Congressus Numerantium 42 (1984), 149–160.

[21] Peter Eades and Xuemin Lin, *Spring algorithms and symmetry*, Theor. Comput. Sci. 240 (2000), no. 2, 379–405.

[22] David Eppstein, Michael T. Goodrich, and Jeremy Yu Meng, *Delta-confluent drawings*, Graph Drawing (Patrick Healy and Nikola S. Nikolov, eds.), Lecture Notes in Computer Science, vol. 3843, Springer, 2005, pp. 165–176.

[23] Martin Erwig, *Inductive graphs and functional graph algorithms*, J. Funct. Program. 11 (2001), no. 5, 467–492.

[24] Jens Fisseler, Gabriele Kern-Isberner, Christoph Beierle, Andreas Koch, and Christian Müller, *Algebraic knowledge discovery using haskell*, PADL (Michael Hanus, ed.), Lecture Notes in Computer Science, vol. 4354, Springer, 2007, pp. 80–93.

[25] Christian Freksa, Wilfried Brauer, Christopher Habel, and Karl Friedrich Wender (eds.), *Spatial cognition iii, routes and navigation, human memory and learning, spatial representation and spatial learning*, Lecture Notes in Computer Science, vol. 2685, Springer, 2003.

[26] Thomas M. J. Fruchterman and Edward M. Reingold, *Graph drawing by force-directed placement*, Softw., Pract. Exper. 21 (1991), no. 11, 1129–1164.

[27] Pawel Gajer, Michael T. Goodrich, and Stephen G. Kobourov, *A multi-dimensional approach to force-directed layouts of large graphs*, Comput. Geom. Theory Appl. 29 (2004), no. 1, 3–18.

[28] Pawel Gajer and Stephen G. Kobourov, *Grip: Graph drawing with intelligent placement*, J. Graph Algorithms Appl. 6 (2002), no. 3, 203–224.

[29] M. R. Garey and David S. Johnson, *Computers and intractability: A guide to the theory of np-completeness*, W. H. Freeman, 1979.

[30] Jeremy Gibbons, *Deriving tidy drawings of trees*, Journal of Functional Programming 6 (1996), no. 3, 535–562, Earlier version appears as Technical Report No. 82, Department of Computer Science, University of Auckland.

[31] ———, *Calculating functional programs*, Algebraic and Coalgebraic Methods in the Mathematics of Program Construction (Roland Backhouse, Roy Crole, and Jeremy Gibbons, eds.), Lecture Notes in Computer Science, vol. 2297, Springer-Verlag, 2002, pp. 148–203.

[32] Bernhard Haeupler and Robert Endre Tarjan, *Planarity algorithms via pq-trees (extended abstract)*, Electronic Notes in Discrete Mathematics 31 (2008), 143–149.

[33] Ivan Herman, Guy Melançon, and M. Scott Marshall, *Graph visualization and navigation in information visualization: A survey*, IEEE Trans. Vis. Comput. Graph. 6 (2000), no. 1, 24–43.

[34] Ralf Hinze and Ross Paterson, *Finger trees: a simple general-purpose data structure*, J. Funct. Program. 16 (2006), no. 2, 197–217.

[35] John E. Hopcroft and Robert Endre Tarjan, *Efficient planarity testing*, J. ACM 21 (1974), no. 4, 549–568.

[36] Paul Hudak, John Hughes, Simon L. Peyton Jones, and Philip Wadler, *A history of haskell: being lazy with class*, HOPL (Barbara G. Ryder and Brent Hailpern, eds.), ACM, 2007, pp. 1–55.

[37] Mark P. Jones, *Functional programming with overloading and higher-order polymorphism*, Advanced Functional Programming (Johan Jeuring and Erik Meijer, eds.), Lecture Notes in Computer Science, vol. 925, Springer, 1995, pp. 97–136.

[38] Simon Peyton Jones, Mark Jones, and Erik Meijer, *Type classes: An exploration of the design space*, In Haskell Workshop, 1997.

[39] T. Kamada and S. Kawai, *An algorithm for drawing general undirected graphs*, Inf. Process. Lett. 31 (1989), no. 1, 7–15.

[40] Andrew Kennedy, *Drawing trees*, J. Funct. Program. 6 (1996), no. 3, 527–534.

[41] David J. King and John Launchbury, *Structuring depth-first search algorithms in haskell*, POPL, 1995, pp. 344–354.

[42] Stephen G. Kobourov and Kevin Wampler, *Non-euclidean spring embedders*, IEEE Transactions on Visualization and Computer Graphics 11 (2005), no. 6, 757–767.

[43] Steffen Mazanek and Mark Minas, *Parsing of hyperedge replacement grammars with graph parser combinators*, ECEASST 10 (2008), 1–14.

[44] Erik Meijer, Maarten M. Fokkinga, and Ross Paterson, *Functional programming with bananas, lenses, envelopes and barbed wire*, FPCA (John Hughes, ed.), Lecture Notes in Computer Science, vol. 523, Springer, 1991, pp. 124–144.

[45] Tamara Munzner, *Exploring large graphs in 3d hyperbolic space*, IEEE Computer Graphics and Applications 18 (1998), no. 4, 18–23.

[46] Bryan O'Sullivan, John Goerzen, and Don Stewart, *Real world haskell*, O'Reilly Media, Inc., 2008.

[47] Benjamin C. Pierce, *Types and programming languages*, MIT Press, Cambridge, MA, USA, 2002.

[48] Bernie Pope, *Re: [fpunion] how useful is lazy-by-default and why?*, Google Groups, The Melbourne Functional Programming Union, July 2009, `http://groups.google.com/group/fpunion/msg/d1a8b32382bd121c`, Accessed August 2009.

[49] Helen C. Purchase, *Which aesthetic has the greatest effect on human understanding?*, Graph Drawing (Giuseppe Di Battista, ed.), Lecture Notes in Computer Science, vol. 1353, Springer, 1997, pp. 248–261.

[50] Aaron J. Quigley and Peter Eades, *Fade: Graph drawing, clustering, and visual abstraction*, Graph Drawing (Joe Marks, ed.), Lecture Notes in Computer Science, vol. 1984, Springer, 2000, pp. 197–210.

[51] N. Quinn and M. Breuer, *A force-directed component placement procedure for printed circuit boards*, IEEE Trans. Circuits Syst. CAS-26 (1979), no. 6, 377–388.

[52] Walter Schnyder, *Embedding planar graphs on the grid*, SODA '90: Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms (Philadelphia, PA, USA), Society for Industrial and Applied Mathematics, 1990, pp. 138–148.

[53] Tom Schrijvers, Simon L. Peyton Jones, Manuel M. T. Chakravarty, and Martin Sulzmann, *Type checking with open type functions*, ICFP (James Hook and Peter Thiemann, eds.), ACM, 2008, pp. 51–62.

[54] W. K. Shih and W. L. Hsu, *A simple test for planar graphs*, Proceedings of the International Workshop on Discrete Math. and Algorithms, University of Hong Kong, 1993, pp. 110–122.

[55] Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine, *The boost graph library: User guide and reference manual (c++ in-depth series)*, Addison-Wesley Professional, December 2001.

[56] Don Stewart, *Math.statistics.fusion*, Web site, January 2009, `http://hackage.haskell.org/packages/archive/statistics-fusion/0.2/doc/html/Math-Statistics-Fusion.html`, Accessed August 2009.

[57] Roberto Tamassia (ed.), *Handbook of graph drawing and visualization*, CRC Press, to appear.

[58] Roberto Tamassia and Giuseppe Liotta, *Graph drawing. in handbook of discrete and computational geometry*, ch. 52, pp. 1163–1186, CRC Press, 2004.

[59] Robert Endre Tarjan, *Depth-first search and linear graph algorithms*, SIAM J. Comput. 1 (1972), no. 2, 146–160.

[60] W.T. Tutte, *How to draw a graph*, Proc. London Math. Soc. 13, 1963, pp. 743–768.

[61] Luca Vismara, Giuseppe Di Battista, Ashim Garg, Giuseppe Liotta, Roberto Tamassia, and Francesco Vargiu, *Experimental studies on graph drawing algorithms*, Softw., Pract.

Exper. 30 (2000), no. 11, 1235–1284.

[62] Philip Wadler and R. J. M. Hughes, *Projections for strictness analysis*, FPCA (Gilles Kahn, ed.), Lecture Notes in Computer Science, vol. 274, Springer, 1987, pp. 385–407.

[63] Colin Ware, Helen C. Purchase, Linda Colpoys, and Matthew McGill, *Cognitive measurements of graph aesthetics*, Information Visualization 1 (2002), no. 2, 103–110.