A Framework for Visualizing Hardness Reductions to Grid-based Games

by

Jeffrey David Shen

B.S., Massachusetts Institute of Technology (2015)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2016

Author .....
Department of Electrical Engineering and Computer Science
August 16, 2016

Certified by .....
Erik Demaine
Professor
Thesis Supervisor

Accepted by .....
Christopher Terman
Chairman, Masters of Engineering Thesis Committee

A Framework for Visualizing Hardness Reductions to Grid-based Games

by

Jeffrey David Shen

Submitted to the Department of Electrical Engineering and Computer Science
on August 16, 2016, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Hardness proofs for grid-based games often use gadgets connected together to represent computational problems. We present an open-source framework to implement these reductions, producing actual game instances out of hard computational instances. Our framework first converts the input problem instance into a graph, then draws the graph in an integer grid (a kind of orthogonal graph drawing problem), and finally replaces nodes and edges in this layout with gadgets. To ensure that the final output is aligned, we use linear programming to constrain how gadgets connect. We apply this framework to Circuit SAT and use it to show examples of reductions to Akari and Minesweeper. Lastly, we describe possible future optimizations to the framework to make the output smaller and how to extend it for a wider variety of games.

Thesis Supervisor: Erik Demaine
Title: Professor

Acknowledgments

I would like to thank my thesis advisor Prof. Erik Demaine for his guidance throughout the project and whose class, 6.890 Algorithmic Lower Bounds: Fun with Hardness Proofs, originally inspired it. I would also like to acknowledge Kevin Wu, who worked with me on the joint class project that would become the starting point of this thesis. Finally, I would like to thank all my family and friends for their continual support throughout this project and my time at MIT.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this thesis, we develop a framework for implementing a variety of reductions from hard computational problems to grid-based games and puzzles. Our goal is to have a framework that can provide implementations of reductions for as many games as possible, while also generating outputs that are visually instructive. That is, the game representations of a hard computational problem should be as small and human-readable as possible. We also end up looking at and providing a solution for an alignment problem in the area of orthogonal graph drawing, where we allow both vertices and edges to have arbitrary rectangular sizes and require them to align at specific points on their boundary.

In this chapter, we give brief backgrounds on reductions to grid-based games and on orthogonal graph drawing. Then, we give an overall overview of the general framework and outline the rest of this thesis.

1.1 Reductions to Grid-based Games

In the area of combinatorial game theory, there has been a large amount of study on the computational complexity of games and puzzles. Primarily, a number of results have been given regarding the hardness of numerous games, including Akari [28], Minesweeper [24], Candy Crush [39, 15], Block-pushing or sliding games [8, 7, 19], ShakaShaka [9], Nintendo games [1], Chess [11], Go [32], and Conway's Game of Life

[30]. However, of these, the only hardness reductions that have been implemented is the one to Candy Crush, which was done by Gualà, Leucci, and Natale [15], and to a variant of block pushing called Push-* done by Oines [29].

All of these reductions and mostly all reductions to games on rectangular grids involve placing down and connecting gadgets that represent components of a computational problem, like variables, logic gates, clauses, or vertices. For example, in the proof that Akari, also known as Light Up, is NP-complete, rectangular gadgets are found that represent variables, wires, turns, crossovers, a splitter, a NOT gate, and an OR gate. The goal of the puzzle is to light up all white tiles by placing light bulbs at specific locations under some constraints, and each gadget in the game only has a certain number of valid solutions corresponding to the part of a circuit they represent. Thus, mapping a problem instance to an instance of the puzzle only requires connecting gadgets in a proper configuration, ensuring that they all align correctly.

Other games, however, have different requirements. In ShakaShaka, the reduction is from planar 3SAT, so no crossover gadgets are provided. Some of the NP-hardness proofs for Nintendo games require the gadgets to be arranged so that a player can traverse all the variables in order, and then all the clauses in order. Other games, like Spiral Galaxies puzzles [12], may not have turn gadgets at all, and instead circuits proceed in one direction with gadgets to allow wires to crossover and gadgets to shift the wires up or down. Lastly, quite a few games may have some gadgets which are non-local in nature. For example, in one class of block pushing games called Push-*, variables and clauses must be arranged in a very specific way, since all blocks are moveable and the player has the ability to push arbitrarily large rows or columns of blocks. Winning condition gadgets may also be somewhat non-local, as they sometimes must provide the player who completes a goal with a very large material advantage to compensate for losses incurred from the rest of the board. These include generalized versions of games like Amazons [17], Chess, or Go.

Given the wide variety of types of games and proofs, it is impossible to develop a framework that covers every single one. As a result, we focus mainly on games like Akari that are reducible from SAT and have connecting gadgets composed of wires,

crossovers, and turns. However, we also describe in this paper how the framework could be extended to cover additional games and the limitations of the framework.

1.2 Orthogonal Graph Drawing

Given that most reductions to grid-based games involve placing down and connecting gadgets, the problem of building a framework that does this well falls under the area of orthogonal graph drawing. In orthogonal graph drawing, graphs must be embedded in a plane with the edges as polylines of horizontal and vertical line segments. From some results in this field, we already have some limitations on the optimality of any such framework, since many possible optimality goals are themselves NP-hard. For example, minimizing the area of a planar embedding of a planar graph in an integer grid [10], minimizing the number of crossover gadgets (crossings) [13], and minimizing the number of turn gadgets (bends in edges) [14] are all NP-hard.

Of course, despite the hardness of orthogonal drawings of graphs, there has been a wide amount of study in tackling it [4]. The topology-shape-metrics approach is widely used and occurs in several phases involving trying to minimize crossings, then bends, and finally area [37]. Note that in the traditional formulation, vertices are points on integer grids and edges are themselves segments along the grid, but others have also considered rectangular nodes with a size. For example, Battista, Didimo, Patrignani, and Pizzonia used network flow techniques to draw graphs where nodes have prescribed sizes and edges are allowed to connect anywhere [3]. Furthermore, Klauske, Schulze, Spönemann, and Hanxleden considered graphs where nodes not only have a certain sizes but edges are also only allowed to connect to specific areas on the node, called ports [26].

However, no approach so far has considered the problem where the edges also are allowed to have arbitrary rectangular sizes and where every part must connect at designated ports. For our use case, many reductions in fact need edges that are composed of rectangular gadgets that include wires, turns, and crossovers. In the course of developing our framework, we give a fairly optimal way to convert from a

given orthogonal drawing of a graph where nodes have size and designated ports to an actual placement of all the gadgets using linear programming.

1.3 Overview of the Framework and Paper

A good deal of work is required to transform the various inputs and outputs of the framework into formats that are usable for the orthogonal graph drawing and by the end user respectively. So, there are quite a few steps in the framework, summarized in Figure 1-1, which are generally divided into pre-processing, orthogonal graph drawing, and post-processing. In the context of the framework, we refer to each of these steps as a module, and these can be chained together to produce an implementation of a reduction in a flexible way. All of the code for the project is hosted on a Github repository [34].

In the pre-processing component, we take an input problem and gadgets, and give a graph representing the input problem which is to be drawn orthogonally. Here, we have a SAT parsing module for parsing a boolean expression, which outputs a directed acyclic graph, which we call an *configuration*. Note that not all reductions necessarily have AND, NOT and OR gadgets, but are only required to have gadgets which are able to construct any such boolean statement. Thus, we also have a configuration substitution module to perform substitutions until our configuration is broken down into gadgets. In this component, we also parse gadgets, as well as other data, like files for the possible configuration substitutions and for the images required for post-processing.

The orthogonal graph drawing component is then composed of two main steps. First, the gadget planarization module generates a general orthogonal layout of the input configuration. The output represents nodes as rectangles with input and output ports, and connecting gadgets like wires, turns, and crossovers as rectangles with unit thickness. This representation is denoted as a *cell configuration*. Then, the gadget alignment module takes the general layout and returns a *gadget configuration*, which stores the locations of all the gadgets to placed. This module fulfills the role of

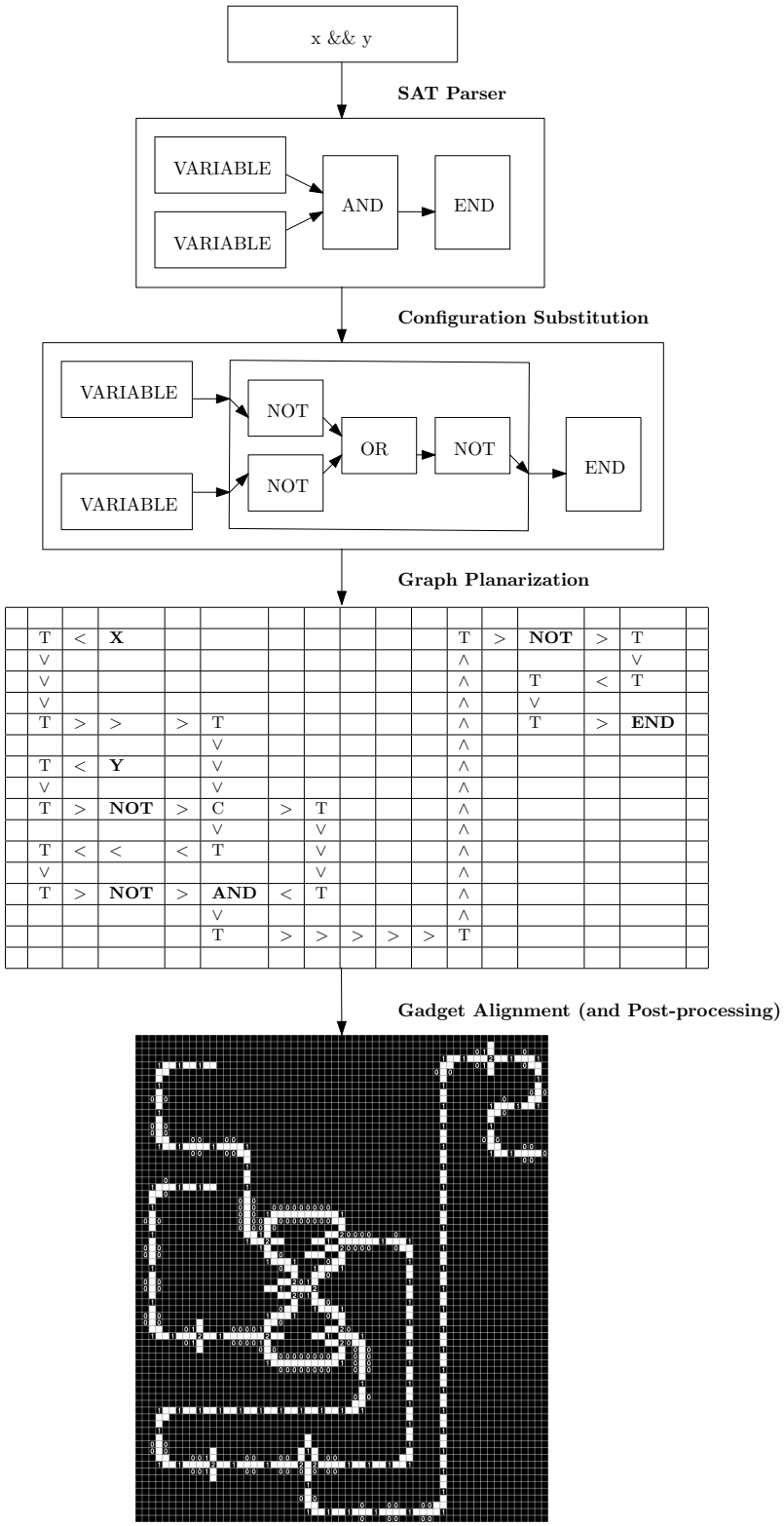


Figure 1-1: A diagram of the overall flow of the framework.

transforming a more traditional orthogonal graph drawing to one with all components composed of rectangular gadgets that have varying sizes.

In the post-processing component, we convert the internal representations used by the framework into more final formats. The gadget placement module takes the gadget configuration and simply returns the result of placing those gadgets onto a grid. That is, this output is simply a grid of the game's tiles, with no additional information about which groups of tiles represent a gadget. An image post-processing module takes a grid of game tiles and outputs an image as a visualization of the game state.

Much of the work on the basic framework came out of a joint class project with Wu [35]. This thesis mainly involved several improvements of the basic framework, but the largest improvement was optimizing the gadget alignment step using linear programming. In Chapter 2 of this paper, we first give a specification of the framework, including all the file formats and modules. In Chapter 3, we describe steps from the basic framework, including the original substitution, planarization, and alignment steps. In Chapter 4, we describe the optimizations made to the planarization and alignment steps. Of particular note in this section is a description of the linear programming constraints involved in aligning all the gadgets. In Chapter 5, we provide some examples and analysis of the performance of the framework using the games for Akari and Minesweeper. In Chapter 6, we describe potential future optimizations to the framework and extensions to additional classes of games. In Chapter 7, we conclude this work.

Chapter 2

Specifications and Formats

The input to our reduction system (specified on the command line) is a single XML file that gives the configuration for the reduction. We first give a specification of the reduction configuration file format, as well as all other file formats and types that are used by the program. These file formats are designed primarily to be human-readable. Then, we also describe all the modules, their inputs and outputs, and their limitations.

2.1 Formats and Types

2.1.1 Reduction Configuration

A reduction configuration file is an XML file that exactly specifies how to produce a reduction. It consists of a data portion and a modules portion. The data portion is usable by all the modules specified in the reduction, and the modules are all run in order, with either the specified input, if present, or the output from the last module fed as the input to the current module. Each module can also export their output as a file.

The root element of the file should be named `reduction` and should have the following structure. Note that repeated denotes an element that can be present any number of times and optional denotes an element that may or may not be present.

<data>
  <gadgets>
    <gadget> (repeated)
      <input type="file" or "string" or "default_resource">
        Value: A file location, string input, or location of a default resource.
      <type> (optional)
        Value: The type of the gadget.
      <symmetries> (optional)
        Value: all or rotation.
  <configs>
    <config> (repeated)
      <input type="file" or "string" or "default_resource">
        Value: A file location, string input, or location of a default resource.
  <images sizeX=(X size) sizeY=(Y size)>
    <image> (repeated)
      <key>
        Value: A string specifying a unique key.
      <input type="file" or "string" or "default_resource">
        Value: A file location, string input, or location of a default resource.
<modules>
  <module> (repeated)
    <name>
      Value: The name of the module to be run,
    <input type="file" or "string" or "default_resource"> (optional)
      Value: A file location, string input, or location of a default resource.
    <output type="file"> (optional)
      Value: A file location.

Here, default resources are files that are bundled with the program, which cur-

rently include several common configuration substitutions. In the data that is used to initialize each module, typed and untyped gadgets are separated and typed gadgets are grouped by their string value. Furthermore, gadgets with a specified symmetry have all symmetries automatically generated with the same name. Lastly, the images are scaled to the given X and Y size, and stored as a map from a key to an image.

2.1.2 Gadget

A *gadget* is a grid of game cells, which are represented by strings, and specified input and output ports on the boundaries of the cell. The file format of a gadget is as follows:

1. A line with the name of the gadget
2. A line with the width x and height y separated by whitespace.
3. $y + 2$ lines with $x + 2$ cells separated by whitespace. The boundary of the grid is either `I_` followed by an input number, `O_` followed by an output number, or `_` if neither. The corners must be marked `_`.

A valid example is given in Figure 2-1.

```
CROSSOVER
1 1
 _  I_0  _
I_1  C  O_1
 _  O_0  _
```

Figure 2-1: An example file for a CROSSOVER gadget.

2.1.3 Configuration

A *configuration* is a named directed acyclic graph with an labeled nodes, with two special nodes specified for inputs and outputs. Furthermore all inputs and outputs are numbered starting at 0, separately, for each node. The input node and the output node can only have outgoing edges and ingoing edges respectively and represent the

input and output of the configuration as a whole. The file format of a configuration is as follows, with empty lines ignored:

1. A line with the name of the configuration

2. A line starting with `input`, followed by a unique positive integer id, and the number of inputs, all separated by spaces.

3. A line starting with `output`, followed by a unique positive integer id, and the number of outputs, all separated by spaces.

4. One or more lines starting with `node`, followed by a unique positive integer id, the name, the number of inputs and the number of outputs, all separated by spaces.

5. One or more lines declaring the outgoing edges for each node in topological order. Each line contains the id of the node, followed by a space and a space-separated list of comma-separated pairs, which can be surrounded by parentheses, with the id of the destination node and the corresponding input number of the edge at the destination node. The input and output node should always be first and last, respectively, and the output node may be omitted.

A valid example is given in Figure 2-2.

```
NAND
input 0 2
output 4 1
node 1 AND 2 1
node 2 NOT 1 1

0 (1,1) (1,0)
1 (2,0)
2 (4,0)
```

Figure 2-2: An example file for a configuration producing NAND from AND and NOT.

2.1.4 Atomic Configuration

An *atomic configuration* is a configuration, a substitution map from strings to other configurations, and a set of string atoms. An atomic configuration is also a directed acyclic graph like a configuration, except where logically equivalent substitutions have been made for various nodes until all the lowest level nodes are atomic. No file format is currently provided for an atomic configuration.

2.1.5 Cell Configuration

A *cell configuration*, which we sometimes refer to as a *layout*, is a grid of layout cells representing a planar embedding of an atomic configuration. Each cell consists of a type, which is one of `EMPTY`, `WIRE`, `TURN`, `CROSSOVER`, `NODE`, `PORT` and numbered input and output directions. If the cell is a port or a node, then it also has a name and a id, which is a list of integers representing its position in an atomic configuration. If a cell is a port, then it also has a specified port number corresponding to each direction. Note that each rectangular block of nodes and ports, which should not touch any other nodes or ports, represents an atom in an atomic configuration, and are connected using wires, turns, and crossovers. Furthermore, note that the port number has a meaning relative to the whole rectangular block, and not an individual port.

To make file format more readable, the beginning of the file declares several macros for the nodes and ports. These are then substituted in for the layout cells where they appear in the grid. The file format for a cell configuration is the following:

1. A line with the number of macros.
2. Zero or more macros, starting with `NODE`, then a unique macro string, a name, and list of integers representing the id, all separated by whitespace.
3. A line with the width, x, and height, y, separated by whitespace.
4. $3y$ lines of $3x$ strings separated by whitespace. Each 3 by 3 block represents a layout cell. The first letter of the center of each layout cell denotes the type of

the cell, which should have the same first letter. If the layout cell is a port or node cell, then the rest of the string denotes the macro to use. For each string cell occupying a cardinal direction from the center, the characters `^`, `V`, `>` and `<` are used to represent the direction north, south, east, and west, and if the first character is the same as the direction the cell is from the center, it is an output, otherwise if it is the opposite, it is an input. The character is afterwards repeated a number of times corresponding to the index of its input or output. Finally, if the layout cell is a port, the rest of that string ends with the port number. If a string cell is none of these things, then `_` is used as a placeholder.

A valid example is given in Figure 2-3.

```
4
NODE 0 VARIABLE 0
NODE 1 VARIABLE 1
NODE 2 OR 2
NODE 3 END 5
5 3
_ _  _  _ _ _  _  _  _  _ _ _ _  _  _
_ P0 >0 > W > >>0 P2 <1 < W < <0 P1 _
_ _  _  _ _ _  _  V0 _  _ _ _ _  _  _
_ _  _  _ _ _  _  V  _  _ _ _ _  _  _
_ E  _  _ E _  _  W  _  _ E _ _  E  _
_ _  _  _ _ _  _  V  _  _ _ _ _  _  _
_ _  _  _ _ _  _  V0 _  _ _ _ _  _  _
_ E  _  _ E _  _  P3 _  _ E _ _  E  _
_ _  _  _ _ _  _  _  _  _ _ _ _  _  _
```

Figure 2-3: An example file for a cell configuration for an expression with an OR operator.

2.1.6 Gadget Configuration

A *gadget configuration* is a collection of gadgets and their offsets on a grid. It may also be used to contain metadata about the gadgets, such as their atomic configuration ids. No file format is currently provided.

2.1.7 String Grid

A string grid is a grid of game cells representing a certain game state. The file format is simply one or more lines consisting of string cells separated by whitespace.

2.2 Modules

Here we give the specifications for a list of modules, their inputs, and their outputs. The official names and types for each of these can be seen in Table 2.1.

Table 2.1: List of module names, inputs, and outputs

Module Name	Input	Output
SATParser	String	Configuration
ConfigurationSubstitution	Configuration	AtomicConfiguration
GadgetPlanarization	AtomicConfiguration	CellConfiguration
GadgetAlignment	CellConfiguration	GadgetConfiguration
GadgetPlacement	GadgetConfiguration	Grid<String>
ImagePostProcessing	Grid<String>	BufferedImage

2.2.1 SAT Parser

The SAT parser module takes in a boolean expression as a string, and outputs a configuration representing the expression. In the boolean expression, variables must be alphanumeric, and the logical operators for OR, AND, and NOT are represented by `||`, `&&`, and `!`, respectively. In the configuration, these correspond to the names `OR`, `AND`, and `NOT`. Variables correspond to nodes with the name `VARIABLE`, which only has one outgoing edge. If a variable is used multiple times, then they also correspond to some nodes named `SPLIT`, which has one input and two outputs. The whole boolean expression feeds into a node named `END`, which usually requires the whole circuit to be true.

2.2.2 Configuration Substitution

The configuration substitution module takes in a configuration and outputs an atomic configuration. The untyped gadget names from the given data are used as atoms and the configurations from the given data are used as possible substitutions.

2.2.3 Graph Planarization

The graph planarization module takes in an atomic configuration and returns a cell configuration, which we also refer to as a layout. The untyped gadgets are used to model any labeled nodes, and the inputs and outputs must match the nodes in the configuration. The nodes for this step must be provided in topological order, and there must be no inputs to the atomic configuration.

2.2.4 Gadget Alignment

The gadget alignment module takes in a layout and returns a gadget configuration. The required typed gadgets are `wire`, `turn`, `crossover`, and `empty`, which represent their corresponding layout cell. These have the following requirements:

- Wires must be straight.
- Turns must have one input and one output which are not on opposite sides of one another.
- Crossovers have two inputs and two outputs with each pair on opposite sides of each other.
- Empty gadgets must be 1 by 1.
- For wires, turns, and crossovers, gadgets for all possible input and output directions must be given with no duplicates except for wires.
- For crossovers, the indices for the pairs do not matter.

- The greatest common divisor of all the wire lengths for a given direction should be 1.
- The thickness on either side of the wire shouldn't hang over the side of any other gadget it can connect to.
- There should be enough separation between ports on the same side of a node for two adjacent ports to have wires coming out and for either wire to make a turn away from the other.
- The thicknesses on each side of the wire should be relatively balanced, since this module currently treats them as a single thickness, rather than two separate ones.

All gadgets are also assumed to have insulation. That is, any two gadgets should be allowed to be placed such that two non-port sides are directly adjacent to each other without it interfering with the validity of the reduction. Lastly, in the layout given, we have a few requirements. First, no two layout cells should be connected directly without a wire in between. Second, there should also be no "live" ports. That is, all input and outputs should be connected to another output or input port respectively.

2.2.5 Gadget Placement

The gadget placement module takes in a gadget configuration and just outputs a string grid that is the result of putting the gadgets at the corresponding offsets.

2.2.6 Image Post-processing

The image post-processing module turns a string grid into a single image. Each string in a cell is mapped using the key for an image, and stitched together to make a final image, which can be exported as a PNG file.

Chapter 3

Basic Framework

In this chapter, we detail a basic implementation of the framework. The three modules of interest are gadget substitution, graph planarization, and gadget alignment, which are essentially all the necessary components to create implementations of reductions. Some of the algorithms described here are naive, and in Chapter 5, we give details on some of the optimizations that were made to this basic framework. In particular, the graph planarization step had a minor optimization, and the localized alignment portion of the gadget alignment module was replaced using a linear programming approach.

3.1 Configuration Substitution

In the configuration substitution module, we are given an input configuration and asked to output an atomic configuration which is logically equivalent to the original configuration. In addition, we have a collection of gadgets, which can be substituted as indivisible nodes, and some configurations that can be substituted for nodes of the same name.

The algorithm we use in the module is essentially a breadth first search starting from the gadgets and ending at the input configuration. We store a set of all the names that we have reached so far, which is initialized to the set of gadget names, and a mapping from a name to the configuration we chose substitute, which is initially

empty. We also store a map of dependencies, i.e., which names are needed for each configuration. Then, in each round, we go through each configuration whose name has been unreached, and see if it all its nodes have substitutions. If so, then the configuration itself is constructible, so we add it to the set of reached nodes, add it to the substitution map, and record which names it needs in the dependency map. If an additional configuration has been reached, we repeat until we can no longer reach any new configurations. Lastly, we prune the substitution map to only include configurations that we need. That is, we start with the input configuration and recursively go through dependencies and their substitutions until we have found only the substitutions that are needed. Note that we only visit each dependency if it's substitution has not been seen yet. Then we return an atomic configuration with these substitutions.

Since our algorithm is effectively a breadth first search, the depth of the dependency DAG is minimized. The runtime of this algorithm is then $O(kmn)$, where k is the depth, n is the number of substitutable configurations, and m is the maximum number of nodes in each configuration. The size of the resulting dependency DAG, which is stored implicitly, could be on the order of m^k, since each configuration can have m nodes that are then expanded recursively. However, when actually performing a specified reduction, these variables are all considered constants, so the runtime and the size of the dependency DAG are both just $O(N)$, where N is the number of gadgets in the input configuration.

3.2 Graph Planarization

In the graph planarization module, our input is an atomic configuration, and we are required to output a cell configuration, which is an orthogonal drawing of the input on an integer grid. The problem here is similar to a standard orthogonal graph drawing problem with nodes of prescribed sizes and with port constraints. We generally wish to minimize the area of our embedding, as well as crossovers and bends.

The general algorithm we use here is fairly naive. We iterate through each node

in topological order, placing down and linking each node to its inputs using Dijkstra's algorithm for finding shortest paths. When placing down any node, we make sure there exists a larger empty rectangle surrounding the node that strictly contains it. If there is no such rectangle, we increase the size of the grid so there is enough space. After connecting each input port, we expand the grid so that the area of any enclosed region is non-zero. We do this by finding all two adjacent cells that conflict, e.g., two wires that run parallel to each other or two ports that are connected without a wire in between, and insert a row or column between them. This, along with the fact that node, port, turns and crossover cells by themselves cannot enclose any region, ensures that we can always find a shortest path from the node we placed onto the grid.

If we assume that each node has at most a constant number of ports and that we place each node so that the number of turns required is constant, then we claim that the width and height expand by at most a constant amount each round. When placing down a node, we only need to expand the grid by at most its semiperimeter plus 2 to ensure there is enough space. Furthermore, when linking two ports, we only need to expand on either side of a straight segment of the path, so this adds at most $O(T)$ rows and columns where T is the number of turns. So, the width and height expand by at most a constant amount each round. The final grid then has area $O(N^2)$ where N is the number of nodes. Furthermore, the runtime is $O(N^3 \log N)$ since we have at most $O(N^2)$ vertices at each round when running Dijkstra's and since both the placement and the expansion steps can be done in $O(N^2)$.

In practice, we relax some of these assumptions, especially the one about a constant number of turns. Our node placement currently just picks the first location, and the basic measure of distance for Dijkstra's algorithm, before optimization, had weights for both wires and turns.

3.3 Gadget Alignment

In the gadget alignment module, we are given a general layout as a cell configuration and are required to output a gadget configuration with all the precise locations of

the gadgets. Here we first describe how we handle the problem of generating wires of arbitrary lengths, and then give a naive implementation of the alignment module using a localized alignment scheme.

3.3.1 Wiring

Since wires themselves are also gadgets, we need to be able to construct a wire of any length from a given set of wires with prescribed lengths. A directly related problem, finding the maximum integer that can't be expressed as a linear combination with nonnegative integer coefficients of some positive integers $a_1 \leq \cdots \leq a_n$, is generally known as the Frobenius problem. This integer, known as the Frobenius number, exists if and only if the integers have a greatest common divisor of 1. Furthermore, there are numerous upper bounds on the Frobenius number, including $(a_1 - 1)(a_n - 1) - 1$ [6], where a_1 and a_n are the minimum and maximum positive integers respectively.

We can then use dynamic programming to construct any wire of a length greater than the Frobenius number. In each round, we take each a_i and for each number that we've been able to construct, we add a_i to it, and record the coefficients if the new number has not been constructed before. After a_n rounds, we then loop through each constructed integer, of which there are at most a_n^2, and look for at least a_1 consecutive numbers. The Frobenius number is right before the start of this sequence, which exists because of the upper bound. We can then construct any wire larger than the Frobenius number by subtracting a_1 until we have a length that we know how to construct.

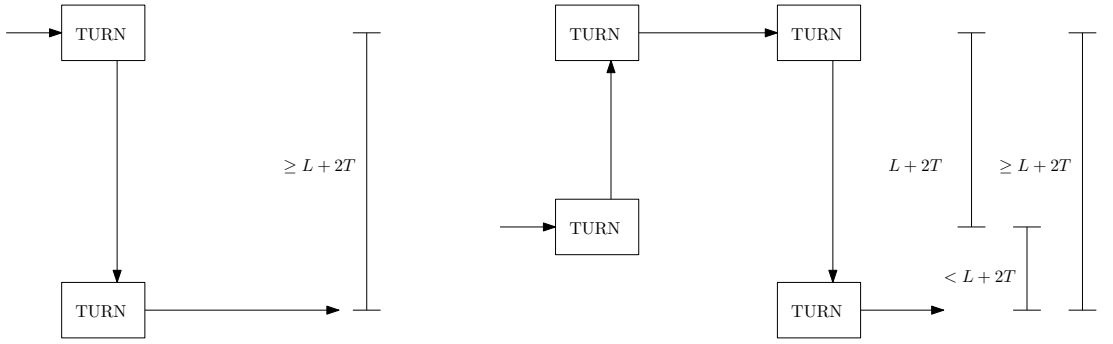
Our algorithm runs in $O(na_n^3)$, since we loop through at most a_n^2 each round. There are more sophisticated algorithms available, yielding runtimes of $O(na_1)$ [5], but practically the current algorithm is enough.

3.3.2 Localized Alignment by Cell

Originally, the alignment portion of the framework used a naive, localized approach. We took each layout cell in the cell configuration and replaced it with a large square

grid of game cells. For any layout cell with an input or output, we forced wires to align locally at the midpoint of the grid.

Note that for wires and turn layout cells, we can simply place down the corresponding gadgets into the grid at the appropriate location. However, for other types of layout cells, with possibly more ports, the general idea here is to use turns to shift any inputs or outputs from a gadget so that the wire is aligned upon reaching the side of the cell. If L is the minimum length of each wire, where we can construct every wire with length at least L, and T is the largest dimension of the turn gadget, then we have two cases. First, if the shift is at least $L+2T$, then we can simply turn twice to reach the proper offset. Otherwise, we have to first shift in the opposite direction, and then shift back, requiring four turns. These two cases can be seen in Figure 3-1.



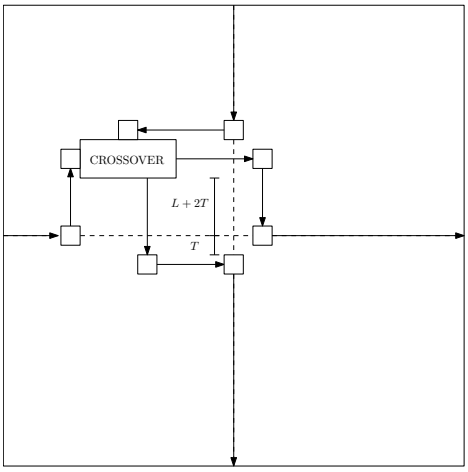
(a) Shifting a wire with two turns when the offset is at least $L + 2T$.

(b) Shifting a wire with four turns when the offset is less than $L + 2T$.

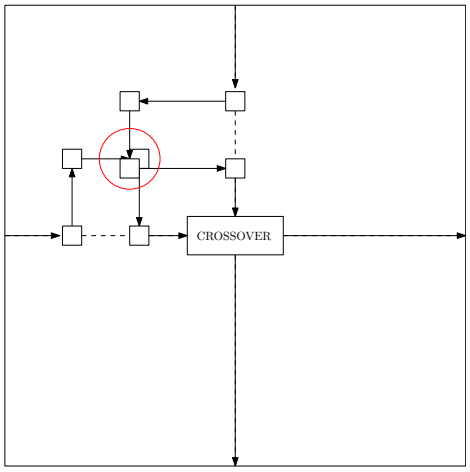
Figure 3-1: The two cases for shifting a wire.

Now, for gadgets with up to a port on each side, like a crossover gadget, the previous scheme may cause some of the wiring to overlap if the protrusions occur in the same quadrant. So, we offset the bottom right corner of the gadget by $L + 2T$ from the center of the cell so that we can use shifts that only require two turns. The proper placement and the potential problem are shown in Figure 3-2. Note that the bottom and right ports must extend to the top-right and bottom-left quadrants using a wire of length $L + 3T$, since otherwise the two turns could still overlap.

Lastly, some gadgets have many ports on the same side, and are represented by a rectangle of layout cells. In this case, we use a rectangle of squares grids to represent the gadget. Similar to the case with a crossover gadget, we offset the top left corner



(a) The bottom-right of the gadget is offset by $L+2T$ from the center both horizontally and vertically.



(b) The bottom right of the gadget is not offset from the center.

Figure 3-2: A proper and improper placement of a gadget with ports on each side.

of the gadget by $L+2T$ from the center. However, since there are multiple turns from the same side, we require that each turn on the same side extends past the last, so they don't overlap. We then stagger the wires. This approach is shown in Figure 3-3. All in all, this adds more requirements to the minimum side length, and we obtain a lower bound of $6L + 2G + 2(P + 3)T$, where G is the maximum gadget dimension, and P is the maximum number of ports on a single gadget side.

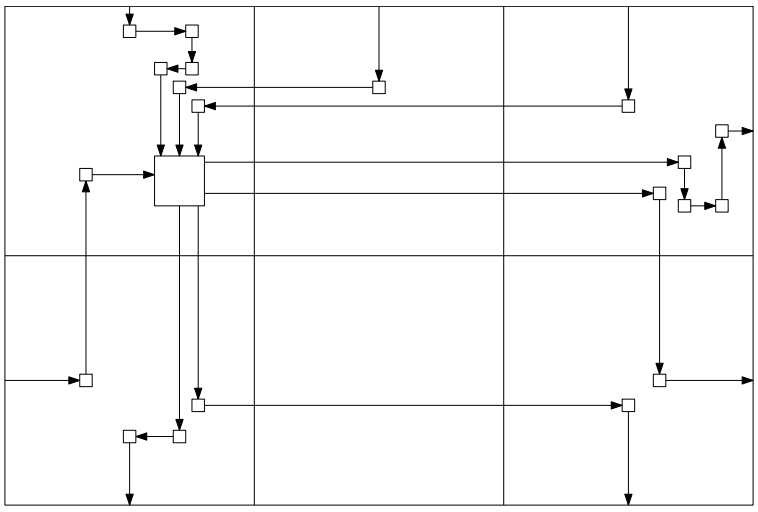


Figure 3-3: A gadget with multiple ports on each side using staggering to avoid overlap.

Chapter 4

Planarization and Alignment Optimizations

In this chapter, we give all the optimizations made to the basic framework. There is a small optimization with the graph planarization step and a major optimization with the gadget alignment step, which replaces the entire module with a linear programming alignment module. For this chapter, we focus mainly on just describing the changes themselves and not the results of the optimizations.

4.1 Graph Planarization

Recall that the planarization step uses a Dijkstra's shortest path algorithm to connect new nodes that are placed down with all its inputs. However, the distance formula originally used only displays a slight bias for straight wires, and doesn't display proper weighting between the importance of avoiding crossovers and turns. So, we adjusted the edge weights so that crossovers were penalized heavily in comparison to turns and wires, with a slight bias towards straight wires.

We also considered and implemented other options with initial placement of the gadget. For example, we tried to place gadgets in locations where the sum of a weighted width and height of the overall cell configuration was minimized. That is, for each row and column, we took the maximum weighted gadget type, with ports and

crossovers penalized heavily, and summed over these values to get a weighted height and width. However, just this additional change, absent any other more sophisticated optimizations, actually worsened performance, so only the previous optimization was included.

4.2 Gadget Alignment

The main optimization of this framework was the gadget alignment step, which was replaced with a module following a linear programming approach. The general idea is that instead of forcing each gadget into a cell and using wiring to make it connect it with the cell boundaries, we can simply straighten out all the wires so that each pair of neighboring gadgets are aligned. Then, the length of the wires must be at least the minimum from the Frobenius problem, and should also be long enough so that none of the gadgets collide. However, these constraints can all be described by linear constraints, and since we can try to optimize the semiperimeter of the output, which is one measure of the size of the output, we can use linear programming.

Note also that for large nodes, an additional submodule is required that shifts a set of inputs and outputs coming from one direction to a new desired alignment. This way, if a node has multiple ports on one side, we can avoid their neighboring gadgets colliding by shifting some of the ports in a direction orthogonal to the direction of the wire.

In the following subsections, we describe all the constraints more rigorously, starting with an overall picture and proceeding from simpler constraints to more complex. We also give a design of the shifter submodule before it's used in the subsection of large node constraints, and a quick correctness argument at the end.

4.2.1 Overall Constraints

For each vertical or horizontal line running along cell boundaries, we define two variables, $\texttt{slicex}_i$ and $\texttt{slicey}_i$ that determine the coordinates of the vertical and horizontal cell boundaries. To prevent gadgets from potentially colliding, the slice variables

are ordered and later constraints will require gadgets to stay inside certain slices. More explicitly

$$\texttt{slicex}_i \leq \texttt{slicex}_{i+1} \tag{4.1}$$

and similarly for the y slices. Our objective function is then just the semiperimeter

$$\texttt{slicex}_n - \texttt{slicex}_0 + \texttt{slicey}_m - \texttt{slicex}_0, \tag{4.2}$$

where m is the size of the cell configuration along the x direction and n is the size along the y direction. For each grid cell, we define eight variables, one for each x and y coordinate of the four sides. These are denoted as $\texttt{sidex}_{x,y,d}$ or $\texttt{sidey}_{x,y,d}$ for the direction d of a cell at (x, y). If there is a port on the given side, then this side variable generally denotes the location of the port corresponding to that side. For any pair of a matching input and output, unless they belong to the same wire, we constrain the two side locations to also be attached to each other in the final configuration. That is, for matching ports at sides (x_1, y_1, d_1) and (x_2, y_2, d_2), where (x_3, y_3) is the vector from the first cell to the second,

$$(\texttt{sidex}_{x_1,y_1,d_1}, \texttt{sidey}_{x_1,y_1,d_1}) + (x_3, y_3) = (\texttt{sidex}_{x_2,y_2,d_2}, \texttt{sidey}_{x_2,y_2,d_2}). \tag{4.3}$$

4.2.2 Wire Constraints

For each wire, with endpoints (x_1, y_1, d_1) and (x_2, y_2, d_2), we only have four constraints. The first is that the wire is straight, i.e., the start and end points align. For example, if the wire is along the y direction, then

$$\texttt{sidex}_{x_1,y_1,d_1} = \texttt{sidex}_{x_2,y_2,d_2}, \tag{4.4}$$

and we have a corresponding equation for the x direction. The second constraint is that the wire has a minimum length, since we can only construct wires above that length via the Frobenius problem. So, if the wire is traveling in the positive y

direction, then we have that

$$\texttt{sidey}_{x_1,y_1,d_1} + \texttt{min\_length} \leq \texttt{sidey}_{x_2,y_2,d_2} \tag{4.5}$$

where `min_length` is the minimum wire length for a wire traveling in that direction with a certain thickness. The cases where the wire is traveling in another direction are similar. Note that different thicknesses can have different minimum lengths, where the minimum thickness is the smallest such that a solution the Frobenius problem still exists. Currently, the desired thickness is set to twice the minimum thickness of the wire.

Lastly, we have two more constraints related to the wire thickness. That is, the boundary of the wire is bounded by the slices on each side of the wire. For example, if the wire is traveling in the positive y direction, then we have the two constraints,

$$\texttt{thickness} + \texttt{slicex}_{x_1} \leq \texttt{sidex}_{x_1,y_1,d_1} \tag{4.6}$$

$$\texttt{thickness} + 1 + \texttt{sidex}_{x_1,y_1,d_1} \leq \texttt{slicex}_{x_1+1}, \tag{4.7}$$

where thickness denotes the desired thickness of the wire. Note that the second inequality requires $\texttt{thickness} + 1$ since it includes one cell for the wire itself. All the other inequalities related to the wire thickness are derived similarly.

4.2.3 Small Node Constraints

For small nodes, which only occupy a single cell, we require one additional pair of variables: `gadgetx` and `gadgety`, which represent the offset of the gadget in the final gadget configuration. Note that this includes both turns and crossovers. Since the ports on the gadget will line up with adjacent wires from Equation 4.3 from the overall constraints, we only need two additional kinds of constraints. First, all ports should have the proper offset from the offset of the gadget. That is, if the gadget is in the cell at location (x_1, y_1) and there is a port on the side in the direction d of the gadget

at the location (x_2, y_2), then

$$(\texttt{gadgetx}_{x_1,y_1}, \texttt{gadgety}_{x_1,y_1}) + (x_2, y_2) = (\texttt{sidex}_{x_1,y_1,d}, \texttt{sidey}_{x_1,y_1,d}), \tag{4.8}$$

Second, the whole gadget must lie inside the slice boundaries. Thus, if the size of the gadget in the x and y directions are `sizeX` and `sizeY`, we have the constraints:

$$\texttt{slicex}_{x_1} \leq \texttt{gadgetx}_{x_1,y_1} \leq \texttt{slicex}_{x_1+1} - \texttt{sizeX} \tag{4.9}$$

$$\texttt{slicey}_{y_1} \leq \texttt{gadgety}_{x_1,y_1} \leq \texttt{slicey}_{y_1+1} - \texttt{sizeY} \tag{4.10}$$

4.2.4 Shifter Submodule

In order to handle large nodes, which occupy a rectangle of cells, we first need a shifter submodule that is able to perform arbitrary orthogonal shifts to wires traveling in a given direction. This way, we can align multiple wires coming out from the same side of a gadget. More precisely, the shifter submodule is able to give a gadget configuration where corresponding pairs of inputs and outputs on opposite sides of a rectangle are wired together, so long as the pairs of ports satisfy certain conditions. To do this in a direction-agnostic way, we label the four directions as in the direction of the shift, in the clockwise direction, in the counterclockwise direction, and in the opposite direction of the shift.

These conditions are the following. First, the ports on the same side are separated by a certain minimum separation. Second, the shifter is allowed to place gadgets within some distance, called the thickness of the shift, in the counterclockwise or clockwise direction of the last ports on both the opposite direction and direction of the shift. The thickness of the shift must be at least some minimum value. Lastly, the length of the shift must be some minimum value. So long as the shift requested has a requested length, thickness, and separations that are at least the minimum values, the shifter submodule can produce a valid wiring.

We now briefly describe how the shift is performed and how the minimum values are calculated. For each port on the opposite side, which is the starting side, we

place wires in the direction of the shift, then turn in the counterclockwise direction, proceed for a bit, turn towards the direction of the shift, proceed a bit, then turn clockwise, proceed a bit, turn back in the direction of the shift, then proceed until the corresponding port in the direction of the shift. These protrusions for each pair of ports are arranged so that each next protrusion surrounds the protrusion of the current one and avoids a collision, as seen in Figure 4-1.

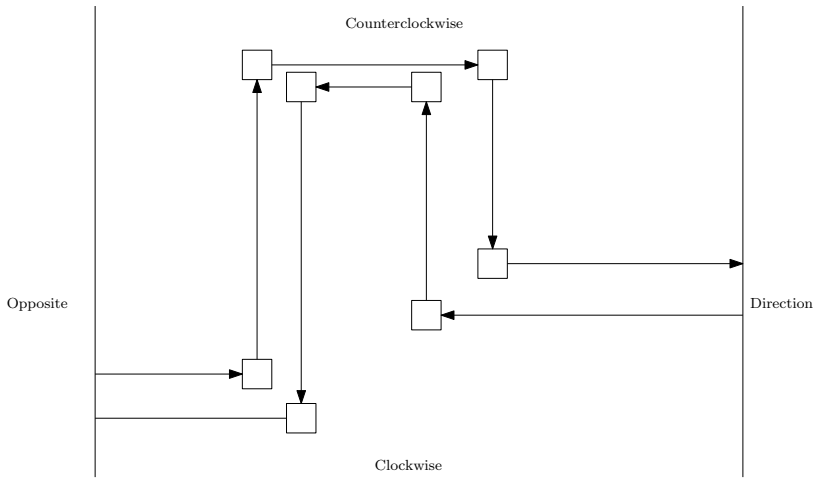


Figure 4-1: The general arrangement of a shift.

For the minimum separation for two adjacent ports on the same side, we only need to avoid the first turn gadget placed for one of the ports overlapping with the wire coming out of the second port.

To calculate the minimum thickness, we first note that it must of course be larger than the thickness of the initial turns of the first ports in the clockwise direction. It must also be larger than the minimum thickness caused by the protrusions in the counterclockwise direction. To calculate this number, we start with the furthest ports in the clockwise direction and calculate the size of the minimum protrusion, which is determined by the minimum length of the wires orthogonal to the direction of the shift. Then we iterate through the ports towards the counterclockwise direction. For each, we note that the protrusion from each port is constrained by both the minimum length of the wire, and also the protrusion of the previous port, since the middle wire cannot overlap any of the turns from the previous protrusion. These constraints are seen in Figure 4-2. In this way, we get how far the last protrusion extends from the

last port, and thus the minimum thickness.

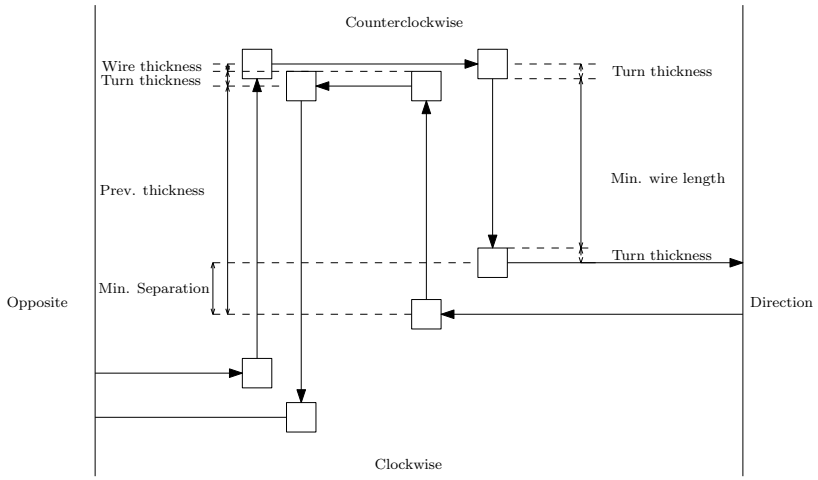


Figure 4-2: The two thickness constraints when iterating.

For the length, we proceed in layers. In the first layer, we iterate through ports from the counterclockwise to clockwise direction. The wire for each port must be such that it is at least the minimum length, and such that the first and second turn are past the turns for the previous port or else they may collide. Similarly, when we reach the middle layer in the protrusion, we iterate in reverse and each wire must go further than the last to ensure that turns do not collide. Lastly, we have that last wire for each port has a minimum length and adding these to the lengths so far of each port, we get the total minimum length. This method is summarized in Figure 4-3.

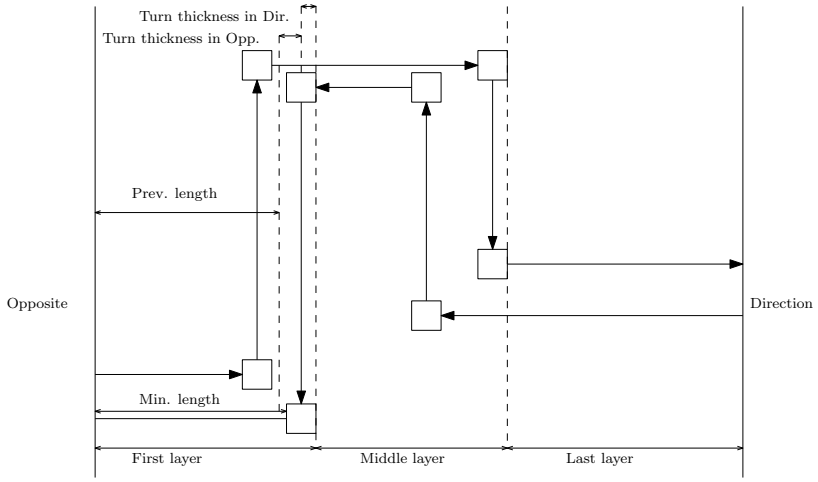


Figure 4-3: The three layers and the length constraints for the first layer.

So long as the shift requested satisfies these minimum constraints, we can then construct a shift by placing down wires and turns in the first and last layer so that each turn is further than the last. Then, we place down the wires and turns for the protrusions, again with each turn further than the last and with the middle wires aligning.

4.2.5 Large Node Constraints

For the large nodes, which are represented by an m by n rectangle of node and port cells, we need a set of constraints for each direction. The idea here is that we can make the gadget larger by extending wires outward from it. Then, since the bounding box becomes larger, we can place shifts on each side of it without worrying about the shifts colliding. This arrangement is seen in Figure 4-4.

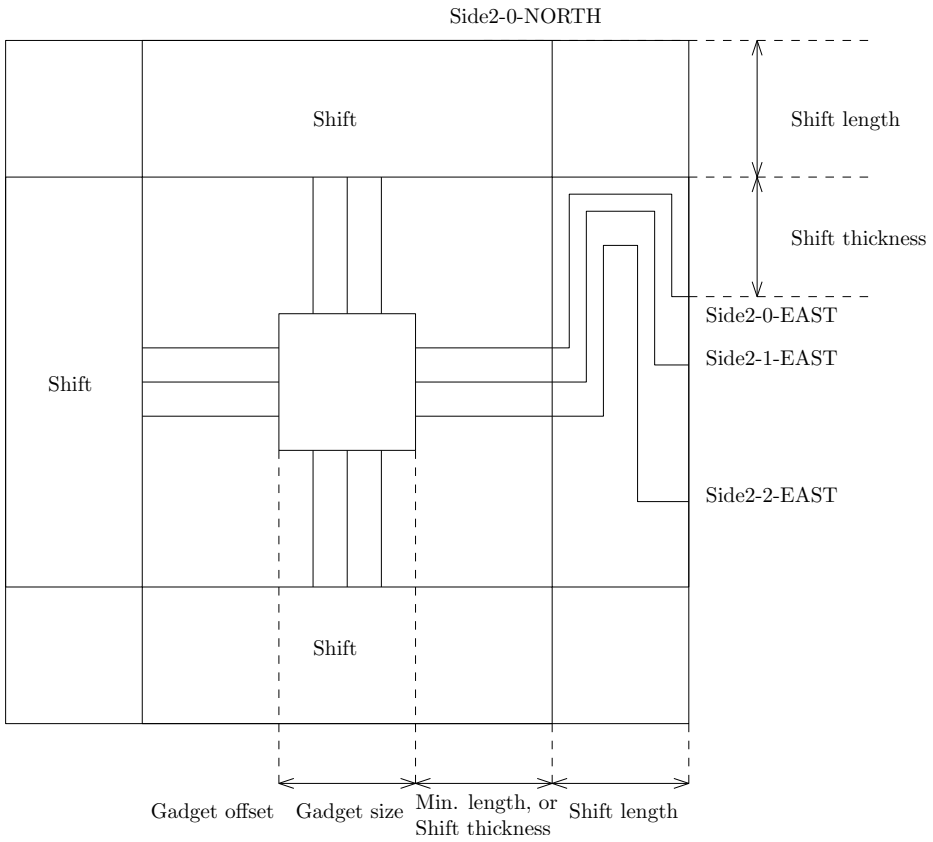


Figure 4-4: The general arrangement of a large node, with a couple constraints from the shift units.

So, we have the following constraints. First, for each direction, all the side vari-

ables for a given direction must be aligned, i.e., they either share the same x coordinate or the same y coordinate. Second, these sides must all be within the cell boundaries given by the slice variables. Next, all the side locations that are ports, must have a minimum separation between them, to satisfy the requirement of the shifter submodule. Afterwards, the two ports on the endpoints for that direction are constrained in the orthogonal direction by the side variables that are in the counterclockwise and the clockwise directions, since those sides and the ports must be separated by a distance of the thickness of one shift plus the length of the other. Note that already, none of the adjacent shifts can overlap, since then the last constraint would be violated.

Lastly, we must be able to place the gadget in the center of the shifts. That is, the boundary of the actual gadget is separated from the boundary of the shift by at least the minimum length of wires in that direction. In terms of the actual variables, the boundary of the actual gadget, which can be expressed as the gadget offset and possibly the size of the gadget, is separated from the respective side by the minimum length of a wire plus the length of a shift. The gadget boundary must also be separated from the shift boundary by the thickness of the clockwise and counterclockwise shifts, so we can simply take the maximum of these. With this last condition, we ensure not only that the shifts don't overlap, but also that we can place the required gadget in the middle.

4.2.6 Overall Correctness

To argue for correctness, we first note that the internal placement of each group of cells is consistent. With small nodes and wires, this is fairly obvious as nodes only have one gadget, and we have a way to construct wiring given the start and endpoints. With large nodes, we already argued that the shifts won't overlap and that there is room for the gadget in the middle. Next, we notice that no two groups of cells will collide, since each is bounded by the slices surrounding them, and because all slices in either direction are ordered, it is not possible for two non-overlapping rectangles of cells to have placements that overlap. Furthermore, we ensured that all ports are aligned, so the whole gadget configuration is aligned properly.

Now, we make an important note which is that this argument may seem to implicitly assume that the solutions to the linear programming problem are all integers. In general, integer linear programming is NP-hard, but here, we simply perform real linear programming and round all numbers to the nearest integer. This preserves correctness since first the length and width of the entire gadget configuration must integers, since otherwise there would be a smaller, integer solution. Second, any gadgets that contribute directly to the objective function will have integer coordinates. Lastly and most importantly, even if a gadget does not contribute directly and so doesn't have integer coordinates, then when we round all numbers, there is no constraint that will be left unsatisfied. The main reason for this is that if two real numbers are separated by at least some integer distance, then upon rounding, they will be still separated by that integer distance. Since all our constraints are of this form or are some equality constraint, rounding gives a valid integer solution. So, using linear programming, we are able to achieve an optimal alignment given the constraints, and a close to optimal alignment in general.

Chapter 5

Examples and Analysis

In this chapter, we discuss the various parts of the system by looking at its output for a variety of different examples. We mainly look at the reduction for Akari [28], but also present examples from Minesweeper [24]. We first look at the linear programming optimization, and compare it against the basic framework. Generally, the linear programming portion produces a close to optimal output given the layout. We then move on to analyze the planarization step, which ends up being the largest contributor to inefficiencies in the final output. Here, we also present a picture of how well the framework could perform against the figure from the Akari reduction paper given further optimization of the layout algorithm. We lastly discuss the shifter submodule and configuration substitution with one last example. Generally speaking, one should avoid multiple ports on the same side of the gadget to avoid incurring the overhead of having to shift the outputs. Furthermore, it is better to manually create gadgets for each logical gate, rather than have the framework perform substitutions.

5.1 Linear Programming

In the basic framework, each cell in the layout was converted independently, so each cell would be very large in order to guarantee alignment with the midpoint of each cell. Even the simple expression for $x \vee y$ was around 1600 rows by 1408 columns in the final output. A zoomed out view of a portion of the final output can be seen in

Figure 5-1(a).

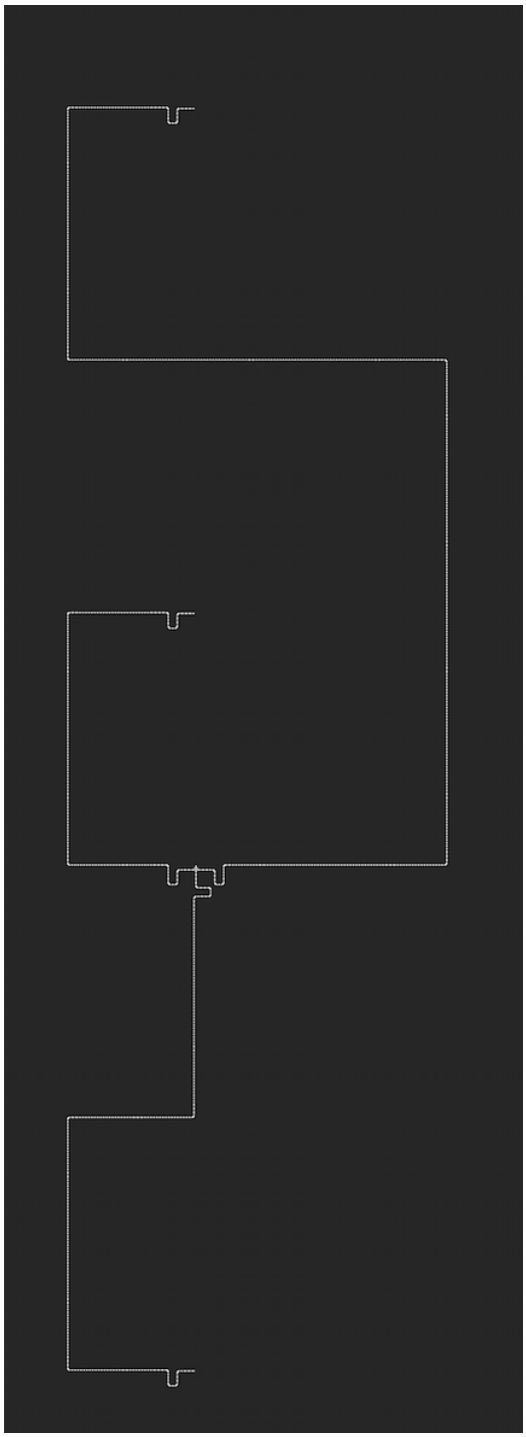
After optimizing the framework using linear programming, alignment becomes handled globally. Thus, the output for the same expression and layout was only 38 by 20 game cells, which is several orders of magnitude smaller. The optimized output for this expression can be seen in Figure 5-1(b), with the input variables near the top and output variable on the bottom. Of course the potential optimal gadget configuration for $x \vee y$ is somewhat smaller, on the order of 10 by 10 game cells or less, since the variable gadgets could be simply rotated and aligned with the ports on the OR gadget. However, for this particular expression, the linear programming produces an output game instance that is very close to the optimal area for the given layout.

5.2 Graph Planarization

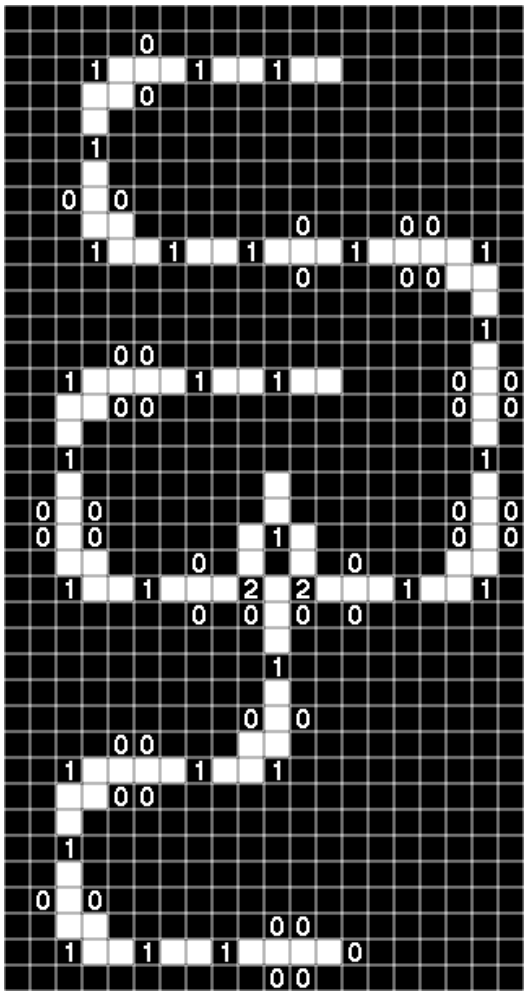
The graph planarization module has remained almost unchanged from the basic framework. However, given the linear programming optimization and the addition of the cell configuration format that allows a user inputted layout, we can now directly look at more complicated expressions where the layout algorithm comes into play.

The example we use is the boolean expression $\neg x \vee ((x \wedge y) \vee z)$ from the Akari paper. With the planarization algorithm, the output of this expression was 134 by 157 game cells, and is presented in Figure 5-2(a). Note that there are several major inefficiencies. First, while the routing is fairly good, there are unfortunately a large number of crossovers, even though the graph itself is planar. This may mostly be since nodes are iteratively placed in the first possible spot, and so crossings are forced to occur if the placement is suboptimal. Furthermore, even if all placements for that node were considered, the placement is still only locally optimal, in that later nodes may be forced into a suboptimal position by a bad earlier placement.

Second, all the major gadgets are not all clustered in the same row or column. From the linear programming step, each contribution to the size by each row or column is determined by the largest cell in that row or column so as to avoid collisions.

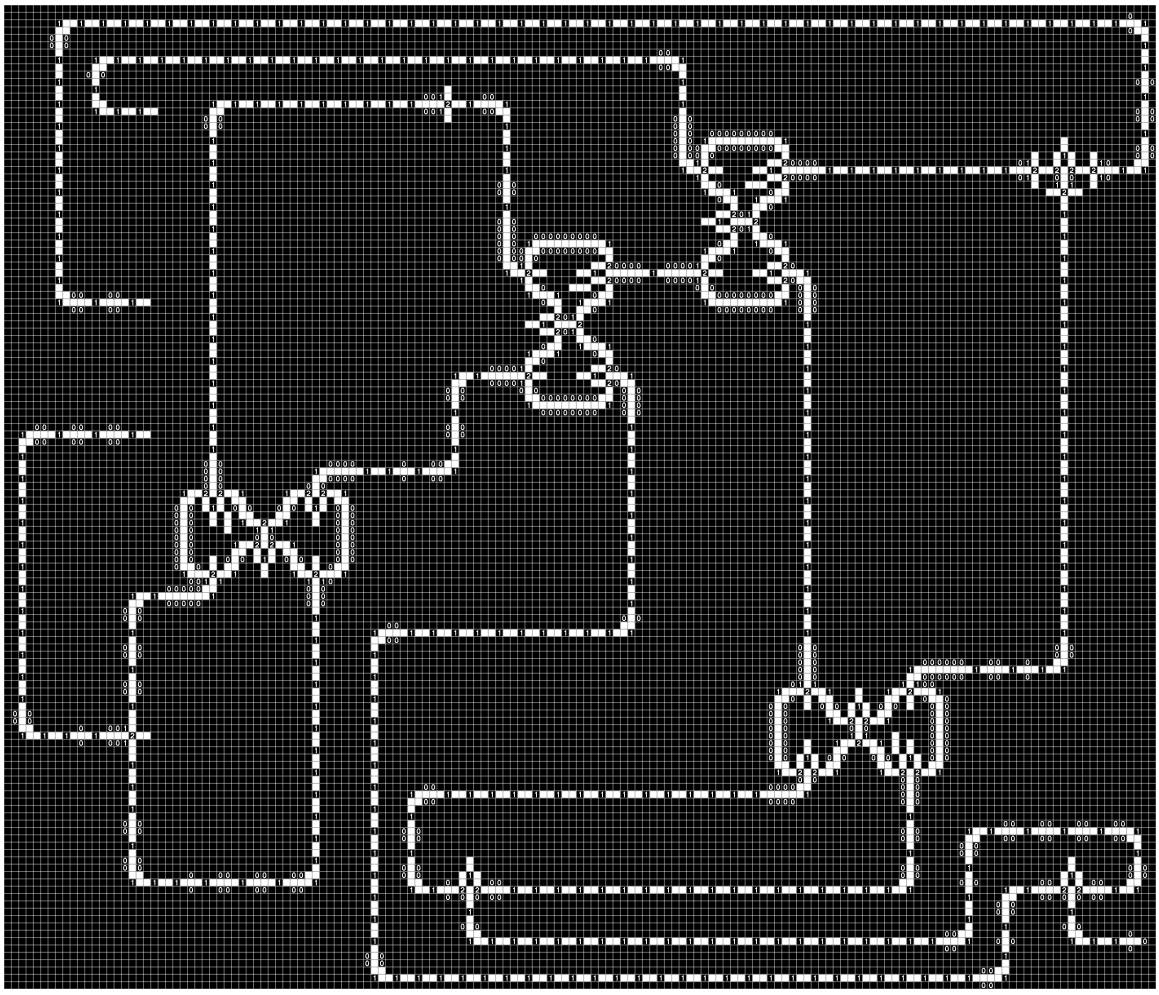


(a) About one-third horizontally of the zoomed-out Akari output instance for the basic framework.

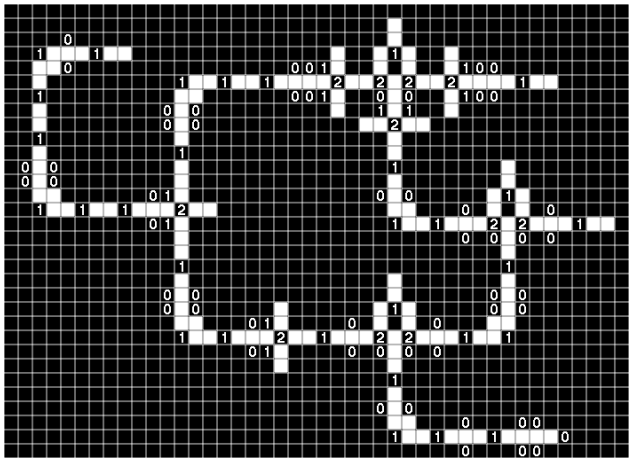


(b) The full Akari output instance for the optimized framework with linear programming.

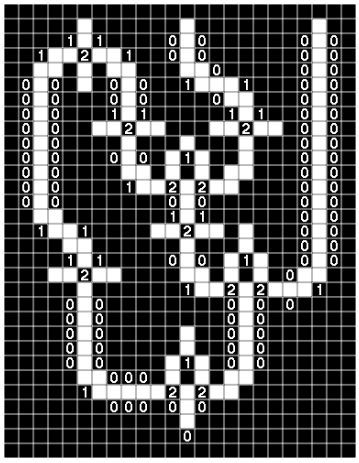
Figure 5-1: A side by side comparison of the outputs for the boolean expression $x \vee y$ of the basic framework and the optimized framework with linear programming.



(a) Output from the layout algorithm.



(b) Output after editing the layout manually.



(c) Instance from the reduction paper.

Figure 5-2: A comparison of the outputs for the boolean expression $\neg x \vee ((x \wedge y) \vee z)$ for the graph planarization algorithm and for a manual redrawing.

Thus, all the crossovers add up, and the final output increases with each additional unnecessary crossover. One optimization we attempted was to choose the placement in a location that already has other gadgets in the same row or column, but this actually made the output worse, since the pathing and placement for later nodes became much worse.

Given this knowledge, we manually redrew the layout, removing crossovers and ensuring that larger gadgets were in the same row or column. This substantially reduced the output size, which then became 32 by 44 game cells. The result is given in Figure 5-2(b). Note that the framework currently only handles variable and logic gadgets in one orientation. For comparison, the original game instance given in the Akari reduction paper is shown in Figure 5-2(c), and is only 31 by 24 game cells. So, we see here that the layout algorithm is the main limiting factor, and given further optimization, we could achieve close to optimal performance.

We present one last example from Minesweeper in Figure 5-3 of the boolean expression $(x \vee y) \wedge (z \vee \neg x)$. The output, after manually editing the layout, is 115 by 91 cells. The unedited layout produced an output of 203 by 287 game cells, which is too large to show here. The output here is much larger for two reasons. First, the gadgets for OR and AND are much larger than the ones from Akari, and second, the wire lengths are 3 and 7, meaning that the minimum wire length is 12, which adds quite a few unnecessarily long wires throughout the gadget configuration.

5.3 Shifter Submodule and Configuration Substitution

Here, we look an example that requires the shifter submodule. Originally, the proof for Akari gave a gadget for SPLIT-NOT, which functioned as both a SPLIT and a NOT gadget. Furthermore, we manually derived the AND gadget from using an OR gadget and three NOT gadgets. Instead, we can have our system create an AND gadget configuration by using configuration substitutions and shifts. The result, with

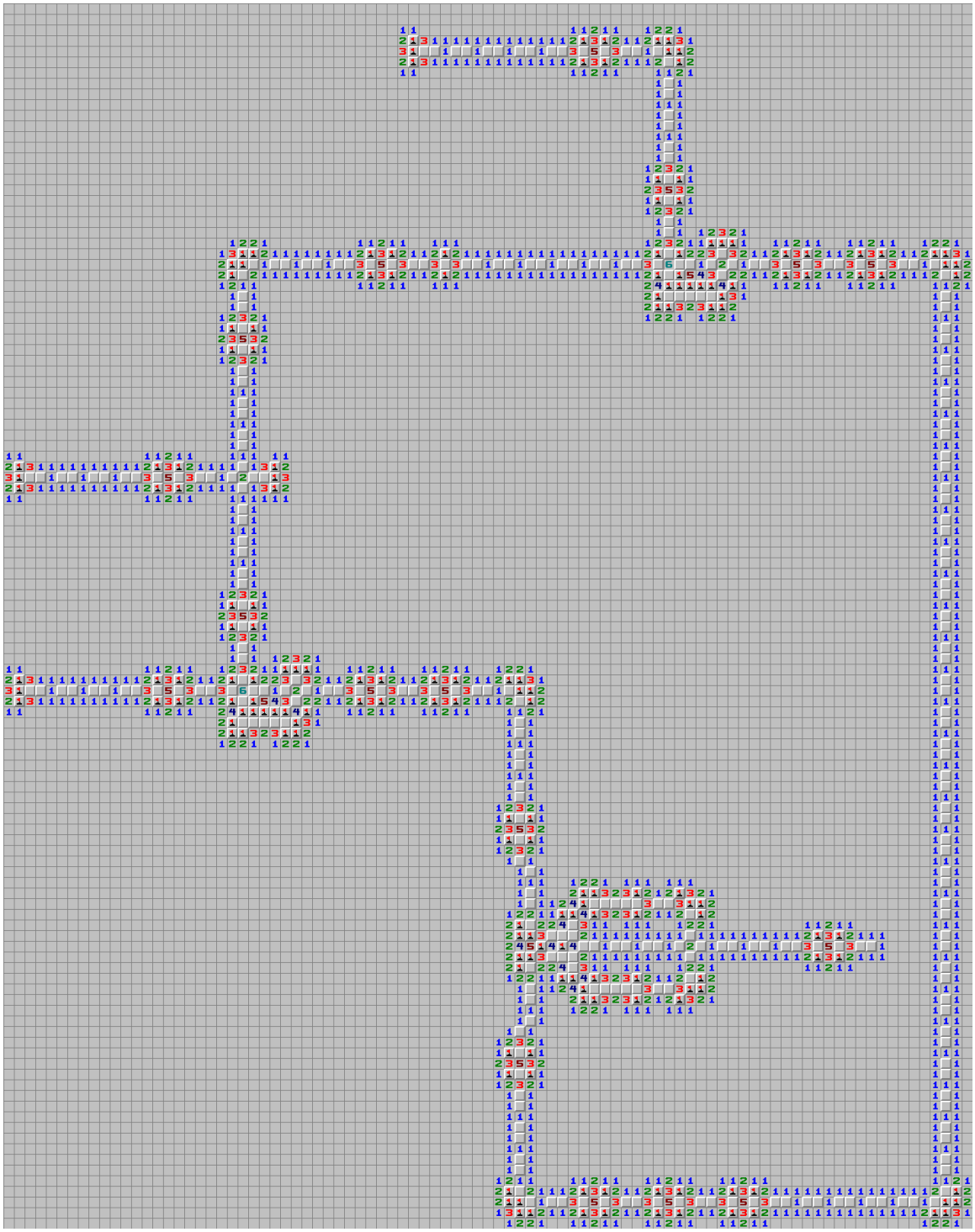


Figure 5-3: The output instance for Minesweeper for $(x \vee y) \wedge (z \vee \neg x)$ with a manually edited layout.

a manually edited layout, is shown in Figure 5-4 and is 85 by 82 game cells. Note that this is incredibly inefficient compared to our derived AND gadget, which is only 10 by 11 cells.

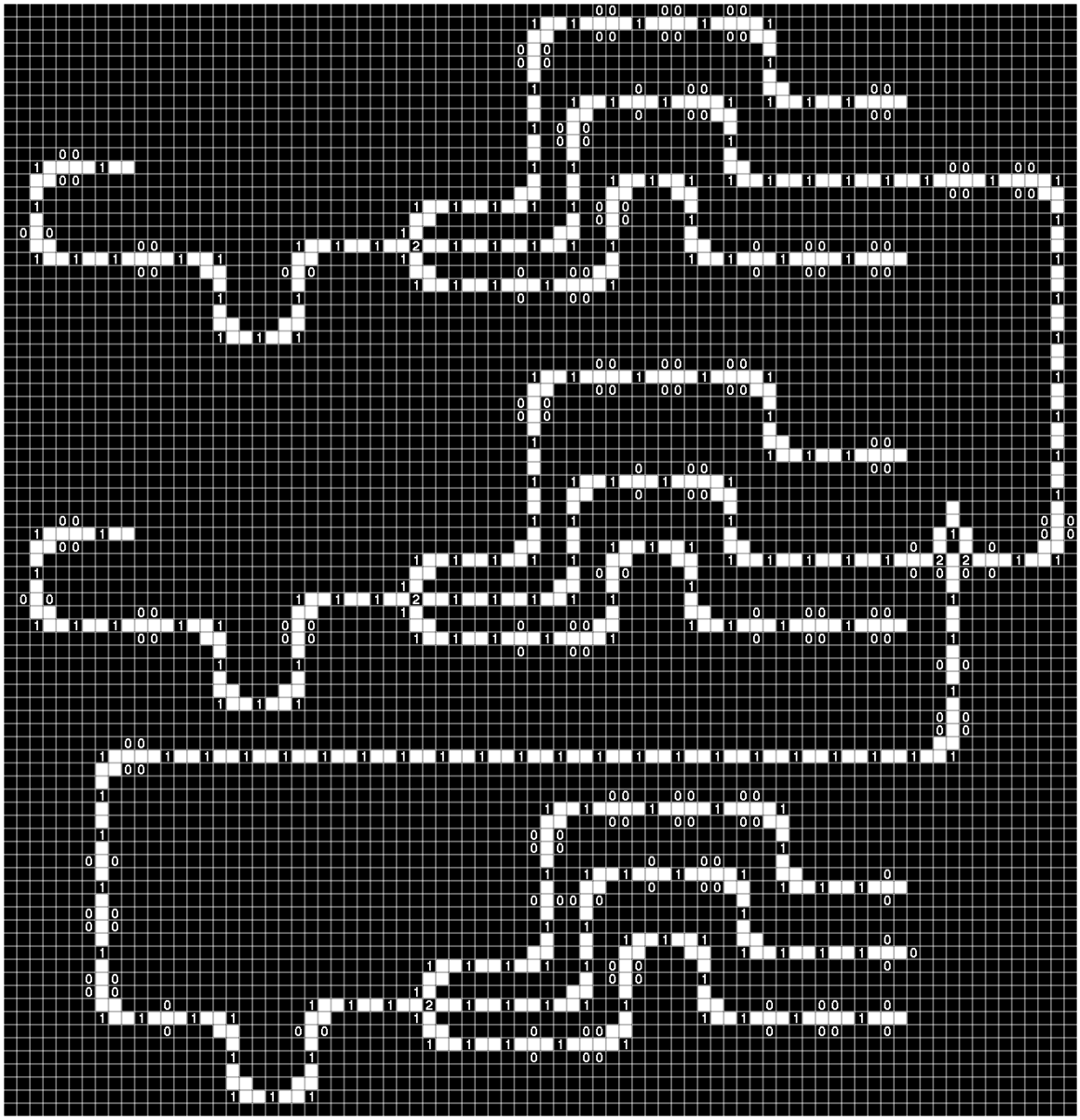


Figure 5-4: Output instance for Akari for $x \wedge y$ with a manually edited layout.

Here, we can see that both the shifter submodule and the configuration substitution module, while necessary to generate all outputs, add inefficiencies to the system. The shifter submodule does so the most in this instance, accounting for a majority of the size both vertically and horizontally. The rest seems to be from using variable and

logic gadgets in a fixed orientation. However, the configuration substitution module still adds some inefficiencies to the system in general, since it makes the graph to draw larger. Thus, the final gadget configuration also becomes larger because of the minimum wire length and because the graph planarization step becomes harder to optimize.

Chapter 6

Possible Extensions and Future Work

There are a variety of directions that this project could take in the future. These directions fall under three main categories. First, there are endless optimizations of the framework we could explore to make the output more readable and instructive. Second, adding generalizations allows the framework to apply to more types of grid-based games and puzzles. Lastly, we could add more features to the framework that enables one to do more with the output for a given game.

6.1 Optimizations

The first and most important area of optimization is in the graph planarization module, which is currently the limiting factor in the compactness of the resulting output. One additional area of optimization is in the gadget alignment step, involving either the shifter, the constraints for large nodes, or some post-processing. The last optimization that one could consider is in the configuration substitution step to account for the size of the atomic configuration rather than the depth, but this is relatively minor.

6.1.1 Graph Planarization

Currently, the graph planarization step is fairly naive, only barely meeting the requirements by placing down and wiring nodes. One optimization may be to repeatedly redraw all the nodes after initial placement by making local improvements, since currently, the planarization step is limited by its single pass approach. Additionally, this could be combined with a heuristic that estimates the size of the resulting string grid by looking at the largest contributor from each row and column. We then select the placement at each step that minimizes the heuristic. We had previously tried an approach with just the heuristic, but as mentioned, this actually increased the size because it made future edge routing harder. However, by combining the heuristic with repeated improvements, we may be able to allow the benefits of using such a heuristic, which is more accurate and directly related to the real size of the final output, to outweigh the drawbacks.

We note, however, that one potential difficulty for this approach may be certain layouts which are local minima but are far away from the global minima. For example, there may be a leaf connected to another node which is connected to its neighbor, but the route goes through a crossover. In the final configuration, the leaf and its connected node should be grouped together and pulled through the crossover, but locally, moving just one of the nodes doesn't improve the configuration at all. So, a lot of configurations, where components of a graph with high connectivity are entangled, become very hard to untangle.

Related to removing and repeatedly placing each node, we could also try a force-directed approach, there are notions of attractive forces between nodes adjacent in the graph and repulsive forces between all pairs of nodes [27]. This is a popular approach that is widely used for drawing graphs. In addition to producing generally good results, especially with fairly small inputs, one advantage of using a force-directed algorithm are that they are relatively flexible with a number of different applications in different fields. Furthermore, a force-directed algorithm would also be fairly simple to implement compared to other graph drawing methods. Some of the drawbacks of a

force-directed approach include a slow runtime and the potential still to be stuck in a local minimum. However, these may be acceptable given that the size of our graph is constrained to be fairly small by the output instances. Force-directed approaches are usually used for non-orthogonal drawings, but can be adapted to orthogonal drawings by minimizing stress and by orthogonalizing the graph in a second step [33, 25].

Lastly, the area of graph drawing in general is well studied, and a number of different layout approaches apply to our orthogonal drawings of directed acyclic graphs. These include hierarchically layered drawing techniques for directed acyclic graphs like the Sugiyama framework [16] as well as the topology-shape-metric approach for orthogonal graph drawing that minimize crossings, then bends, then area [37]. One of the drawbacks, however, of a multiphase approach are that since crossovers and turns are weighted differently in our model, a planarization that aims to minimize crossovers may not be correct when adding in turns or when arranging the crossovers on a grid. Overall, in this module it is important to consider that the layout is constructed from a relatively small configuration and that an additional alignment phase occurs afterwards which gives a slightly different heuristic from the usual.

6.1.2 Gadget Alignment

One of the optimizations involves the shifter submodule, where some of the parts become somewhat large because the wires have a required minimum length. Here, we can compress some of the wires since for some portions, we only require a strictly increasing sequence of wires to ensure that each turn is past the last, not a minimum length. In addition, there are different ways to lay out the shifts that can be explored and that may be more optimal, like having protrusions on both sides. One related improvement is to deal more properly with different wire thicknesses. Currently, both left and right thicknesses are treated the same, when they should be dealt with differently, which may allow some constraints to be more aggressive. Furthermore, most of the code tries to minimize the thickness, but this could also come at the expense of a smaller wire length, so finding the right balance between these aspects is desired.

With the constraints for large nodes, the main optimization here would be to handle cases where only one or some of the sides have multiple ports, since then shifts may not have to be produced for the other sides. This would help especially in the case of split gadgets, where there is only one input, but two outputs, often on the same side.

Lastly, after the gadgets have been aligned and the module has returned a gadget configuration, some post-processing could compact the resulting diagram by looking for sets of wires that could be shortened. For example, we could find a cut of the underlying graph such that the wires in the cut are all parallel, and then see if we can shorten the wires to make the graph smaller, while still keeping all gadgets aligned without collisions. We could also find cuts involving only a single edge and try to shorten the edge by removing wiring in between parts are in the same direction.

6.1.3 Configuration Substitution

In the configuration substitution step, we currently use a breadth first search to find a substitution that minimizes the depth of the resulting dependency graph. However, even though in the worst case the size of the atomic configuration could be exponential in the number of substitutable configurations provided, we can improve the size in cases where minimizing the depth and the size give different results. We could do this by using Dijkstra's algorithm and weighting configurations with what its resulting size would be. So, we would correctly choose atomic configurations which are deeper but have a smaller size.

6.2 Generalizations to Additional Games

Since the general idea of our framework has the ability to be applied to any reduction involving graph drawing, there are a number of different types of reductions we can extend this to. Here we discuss the various modules or components that would have to be added and the difficulty of implementing an extension that covers certain classes of games.

6.2.1 3SAT

For 3SAT reductions, one would have to provide both a variable gadget and a clause gadget. In the most general form, one would in fact have to provide a variety of clause gadgets, one for each combination of positive and negative literals. However, one could easily substitute a positive clause gadgets and negation gadgets for these different clause types.

So, to implement 3SAT would be fairly easy, we only need a 3SAT parser module and some configuration substitutions to make it more usable. However, note that there are a number of possible ways that the satisfiability is checked. One would be that the win condition is a global property with is checked at each clause. However, here, crossovers are not necessarily needed since this falls under planar 3SAT, which is NP-complete, and we cover this in Section 6.2.2.

Another possibility is that the end condition is a single, local gadget, but this requires it to be linked with the clause gadgets. This could occur in a few ways, one of which is covered in Section 6.2.3, where each clause links to the next, a problem which we denote as traversable 3SAT. Another possible way may be to use some number of AND gadgets to connect all the clauses, but here, it seems to be much easier to simply reduce from SAT instead of trying to find clause gadgets.

6.2.2 Planar 3SAT

Of the planar variants of SAT, the most popular formulation is called planar 3SAT, where the graph formed by the variables, clauses, and a loop connecting all the variables is planar [20]. However, we also use this section to discuss the variant where there is no loop, which is also NP-complete. In either case, we will need a 3SAT parser that converts to variable and clause nodes. Furthermore, with the loop variant, then we would need to allow directed graphs with cycles in our graph planarization step. In addition, we need an additional module that is able to create planar embeddings of planar graphs without crossovers. This is possible to do in polynomial time [21]. Furthermore, once the planar embedding is fixed, it is also

possible to find an embedding in a rectilinear grid minimizing bends in polynomial time [36], although minimizing the number of bends when the planar embedding is not fixed is hard to even approximate [14].

The primary example of a game that uses this extension would be ShakaShaka [9], which has a variable gadget, a clause gadget which is locally satisfied only if all inputs are true, a negation gadget, and no loops among the variables. Note specifically that here there is no need for an END gadget like in SAT.

6.2.3 Traversable 3SAT

In some reductions to games, a player must move through all the variables, setting them either true or false, and then can only walk through all the clauses if all of them are satisfied, eventually reaching an end condition. To differentiate these from other possible 3SAT reductions, we denote these reductions as being from traversable 3SAT. Traversable 3SAT reductions cover a wide variety of games, from showing wide number of Nintendo games are NP-hard [1] to showing variants of block pushing games are NP-hard [8].

To implement these types of reductions, we need a traversable 3SAT parser, which generates a configuration with variables, clauses, and an end condition such that variables and clauses have outputs that lead to the next gadget, so that there is a path from the start to the end. In addition, care should be taken with the crossovers, since some games involve unconventional crossover gadgets, where the two wires are ordered and a player cannot traverse the first wire after traversing the second.

6.2.4 Shift Gadgets without Turns

One game whose NP-hardness reduction contains no turns at all is called Spiral Galaxies [12]. Here, all gadgets travel in one direction, and there are shift gadgets instead, which are able to shift an output wire up by one cell, or down by one cell. In this case, both the graph planarization and the gadget alignment steps will require adjustments. First, for the graph planarization step, the layout will must likely have

to be done with layers, and connections between layers would require shifts, which would have occupy at least two layout cells, and crossovers, which would have two inputs on the same side and two outputs on the same side. Second, the gadget alignment step would need a shifter submodule built from shift gadgets and would also need to derive new guarantees for shift layout cells. Here, the guarantees would be interesting since a larger shift would require a greater length but the length is constrained by the size of the shift.

6.2.5 Hamiltonian Cycle

There are a few variants of the Hamiltonian cycle that are used for proofs. In particular, finding a Hamiltonian cycle is still NP-complete if the graph is an undirected planar graph with max degree 3 [22]. The problem is still hard even if the graph is also a grid graph, but these reductions are fairly trivial to implement, like with Slitherlink [40] and Hashiwokakero [2]. With Slitherlink, however, and games that involve single-use paths or tokens and tolls, we have reductions from the first Hamiltonian cycle problem [38]. Here, we require a way to input and deal with undirected graphs, and also need a module that can generate planar embeddings of planar graphs on integer grids.

6.2.6 Quantified Boolean Formulas

Many games, like ones involving doors and pressure plates [38], are PSPACE-hard through quantified boolean formulas (QBF), which in addition to gadgets from either SAT or 3SAT, require quantifier gadgets. Furthermore, the layout may be similar to the traversable 3SAT problem, where quantifier gadgets and clause gadgets may be connected. So, to implement these reductions, we require a parser that will translate a QBF problem into the appropriate configuration, but afterwards, the planarization and gadget alignment steps work the same.

6.2.7 Constraint Logic

Aside from QBF, another way to prove PSPACE-hardness is through nondeterministic constraint logic (NCL) [18]. These involve graphs with colored, directed edges whose directions can be flipped under certain constraints at vertices in order to reach a desired configuration. In particular, planar NCL graphs with only AND and OR vertices is NP-complete. The primary applications here are sliding block puzzle games, like Sokoban or Rush Hour [19].

Again, the main extensions that we require here are a planarization module for planar graphs and a new parser that translates NCL graphs. We also may expect that new wires are required, since they can be colored. However, generally proofs only use the color at the vertices, and wires for either color are exchangeable, so for practical purposes, colored wires are not necessary. Still, to implement colored wires, we would need to add some additional metadata to wires throughout the whole framework.

There are also some games involving bounded 2 player constraint logic (2CL), like Othello [23], Amazons, Konane, and Cross Purposes [17]. Generally, the extensions required are the same as the ones for NCL, but for 2CL, there is oftentimes also a win gadget that can become arbitrarily large. These win gadgets are difficult to handle in a general way, but it may be possible to perform the alignment step so that the victory conditions can be connected to a gadget configuration as a post-processing step.

6.2.8 EXPTIME

Proofs about EXPTIME-hardness primarily include Checkers [31], Chess [11], and Go [32]. Here, the general idea for each of these proofs is that players flip variables trying to satisfy their given formula. However, each of these proofs are very difficult compared to the others, involving a number of different parts, with fairly global win conditions. Furthermore, the general layout of the wiring seems to also be specific to each proof. So, even though these are all grid-based games, the difficulty of the implementations of these games may put them outside the scope of this framework.

6.3 Additional Features

Finally, there are additional miscellaneous features that help to expand the power of the framework beyond just giving the game representation of a problem. The three major ones are using the framework to also produce solved states, creating playable games, and verifying reductions.

6.3.1 Producing Solved States

After performing the gadget alignment step, we can also produce the solution for a given gadget configuration by replacing all the gadgets with their corresponding gadget in the solution to the input problem. However, note that because of possible configuration substitutions, the system in a lot of cases may not know what the actual solved gadgets will be. For example, if an AND configuration is composed of many different gadgets, then we only know what the inputs and outputs to the overall configuration should be, but not to the internal inputs and outputs.

Thus, the user would need to specify for each configuration that they use a corresponding solved state configuration for each of the possible inputs and outputs. We can then find the corresponding atomic configuration. Afterwards, we can use metadata information in the gadget configuration, which has their corresponding id in the atomic configuration, to replace all the gadgets with their solved state counterpart.

6.3.2 Post-processing and Playable Games

Currently, the framework only exports a PNG image file. However, one may want the output in a number of other different formats, like an SVG or an actual playable game instance. Creating new post-processing modules to allow various image formats should be fairly easy and general. For playable game instances, however, we would need create a different post-processing module for each individual game to convert it to a format that a game engine understands.

6.3.3 Reduction Verification

While having this framework output a valid game state strongly suggests that the reduction is correct, it doesn't definitively help verify any particular part of the reduction. Thus, another feature would be a reduction verifier that can check to ensure that the proof provided is actually correct given that the gadgets themselves work as they are intended.

There are a few major aspects to verify. The first is to ensure that the provided atoms and configurations is complete, and can be used to break down any input problem given by the parser. In most cases, this can be done by checking to see if some complete set of components, like AND and NOT for SAT, can be broken down into the provided gadgets. Second, the wire lengths provided should have a greatest common divisor of one. Third, the spacing between ports on the same side of a gadget should be at least some minimum given by the shifter submodule, so that turns can be performed. Lastly, any gadget with ports on two adjacent sides should allow enough space on either side of the port so that wires connected to the gadget don't collide.

In addition, we would also need to make more minor verifications, all of which have been given in Section 2. If a reduction satisfies these checks, then this indicates a correct proof. Note though that the converse is not necessarily true, which is that some correct proofs might fail the checks because they lie outside the bounds of the framework. However, in these cases, we would be able to list the step it failed on, which the user would then be able to manually verify.

Chapter 7

Conclusion

A wide variety of reductions to grid-based games can be done simply by giving an orthogonal drawing that is polynomial in the input graph using a set of given gadgets. However, while the existence of such a drawing is easy, actually finding a drawing is not and requires a great deal of coding. This thesis laid out a path for easily implementing reductions by providing a framework that is capable of drawing graphs using gadgets for all the nodes and edges. In particular, we gave a way to extend current orthogonal graph drawing methods to drawings with these constraints using linear programming. That is, given a general layout of the output, we give a set of constraints that if satisfied, produce a valid output instance with aligned gadgets. Furthermore, our method optimizes for the semiperimeter of the drawing, and we showed that it is possible, perhaps by using better layout techniques from the field, to achieve reduction outputs that are human-readable and efficient. Specifically, we used the system to create specific instances for Akari, also known as Light Up, and Minesweeper.

Lastly, we gave a detailed description of the future work, from optimizations to other games to additional features. Since the linear programming was the main piece from this project, we primarily detail ways to improve other parts of the framework, especially the graph planarization module that gives a general layout. Furthermore, we go into depth about the variety of different games that this framework could cover, and give a theoretical description of how these could be achieved. In conclusion, by

solving a gadget alignment problem, we are able to apply general graph drawing techniques to gadgets and give a general framework that would help people verify, visualize, and play with hardness reductions to grid-based games.

Bibliography

[1] Greg Aloupis, Erik D. Demaine, Alan Guo, and Giovanni Viglietta. Classic Nintendo games are (computationally) hard. In *Fun with Algorithms*, pages 40–51. Springer, 2014.

[2] Daniel Andersson. Hashiwokakero is NP-complete. *Information Processing Letters*, 109(19):1145–1146, September 2009.

[3] Giuseppe Di Battista, Walter Didimo, Maurizio Patrignani, and Maurizio Pizzonia. Orthogonal and quasi-upward drawings with vertices of prescribed size. In Jan Kratochvíyl, editor, *Graph Drawing*, volume 1731 of *Lecture Notes in Computer Science*, pages 297–310. Springer Berlin Heidelberg, 1999.

[4] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, Upper Saddle River, NJ, 1998.

[5] D. Beihoffer, J. Hendry, A. Nijenhuis, and S. Wagon. Fast algorithms for frobenius numbers. *Electronic Journal of Combinatorics*, 12, 2005.

[6] Alfred Brauer. On a problem of partitions. *American Journal of Mathematics*, 64(1):299–312, 1942.

[7] Erik D. Demaine, Martin L. Demaine, Michael Hoffmann, and Joseph O'Rourke. Pushing blocks is hard. *Computational Geometry*, 26(1):21–36, August 2003.

[8] Erik D. Demaine, Martin L. Demaine, and Joseph O'Rourke. PushPush and Push-1 are NP-hard in 2D. In *In Proceedings of the 12th Canadian Conference on Computational Geometry*, pages 211–219, 2000.

[9] Erik D. Demaine, Yoshio Okamoto, Ryuhei Uehara, and Yushi Uno. Computational complexity and an integer programming model of shakashaka. *IEICE TRANSACTIONS on Fundamentals of Electronics, Communications and Computer Sciences*, E97-A(6):1213–1219, June 2014.

[10] Danny Dolev, Tom Leighton, and Howard Trickey. Planar embedding of planar graphs. *Advances in Computing Research*, 2:147–161, 1984.

[11] A.S. Fraenkel and D. Lichtenstein. Computing a perfect strategy for $n \times n$ chess requires time exponential in n. *Journal of Combinatorial Theory (Series A)*, 31:199–214, 1981.

[12] Erich Friedman. Spiral Galaxies puzzles are NP-complete. `http://www2.stetson.edu/~efriedma/papers/spiral.pdf`, 2000.

[13] M.R. Garey and D.S. Johnson. Crossing number is NP-complete. *Journal on Algebraic Discrete Methods*, 4, 1983.

[14] Ashim Garg and Roberto Tamassia. On the computational complexity of upward and rectilinear planarity testing. *SIAM J. Comput.*, 31(2):601–625, February 2001.

[15] Luciano Gualà, Stefano Leucci, and Emanuele Natale. Bejeweled, Candy Crush and other match-three games are (NP-)hard. In *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*, pages 1–8, August 2014.

[16] Patrick Healy and Nikola S. Nikolov. Hierarchical drawing algorithms. In Roberto Tamassia, editor, *Handbook of Graph Drawing and Visualization (Discrete Mathematics and Its Applications)*, chapter 13. Chapman and Hall/CRC, 2013.

[17] Robert A. Hearn. Amazons, Konane, and Cross Purposes are PSPACE-complete. *Games of No Chance 3*, 56:287–306, 2009.

[18] Robert A. Hearn and Erik D. Demaine. The nondeterministic constraint logic model of computation: Reductions and applications. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming*, ICALP '02, pages 401–413, 2002.

[19] Robert A. Hearn and Erik D. Demaine. PSPACE-completeness of sliding-block puzzles and other problems through the nondeterministic constraint logic model of computation. *Theoretical Computer Science*, 343(1-2):72–96, October 2005.

[20] Robert A. Hearn and Erik D. Demaine. *Games, Puzzles, and Computation.* A. K. Peters, Ltd., Natick, MA, USA, 2009.

[21] John Hopcroft and Robert Tarjan. Efficient planarity testing. *Journal of the ACM*, 21(4):549–568, October 1974.

[22] Alon Itai, Christos H. Papadimitriou, and Jayme Luiz Szwarcfiter. Hamilton paths in grid graphs. *SIAM Journal on Computing*, 11(4):676–686, November 1982.

[23] Shigeki Iwata and Takumi Kasai. The Othello game on an n × n board is PSPACE-complete. *Theoretical Computer Science*, 123(2):329–340, January 1994.

[24] Richard Kaye. Minesweeper is NP-complete. *The Mathematical Intelligencer*, 22:9–15, 2000.

[25] Steve Kieffer, Tim Dwyer, Kim Marriott, and Michael Wybrow. HOLA: Human-like orthogonal network layout. *IEEE Transactions on Visualization and Computer Graphics*, 22(1):349 – 358, January 2016.

[26] Lars Kristian Klauske, Christoph Daniel Schulze, Miro Spönemann, and Reinhard von Hanxleden. Improved layout for data flow diagrams with port constraints. In Philip Cox, Beryl Plimmer, and Peter Rodgers, editors, *Diagrammatic Representation and Inference*, volume 7352 of *Lecture Notes in Computer Science*, pages 65–79. Springer Berlin Heidelberg, 2012.

[27] Stephen G. Kobourov. Force-directed drawing algorithms. In Roberto Tamassia, editor, *Handbook of Graph Drawing and Visualization (Discrete Mathematics and Its Applications)*, chapter 12. Chapman and Hall/CRC, 2013.

[28] Brandon McPhail. Light Up is NP-complete. `http://www.mountainvistasoft.com/docs/lightup-is-np-complete.pdf`, 2005.

[29] Asa Oines. Visualizations of block pushing hardness reductions. From 6.890 Algorithmic Lower Bounds: Fun with Hardness Proofs, a course at MIT, 2014.

[30] Paul Rendell. Turing universality of the Game of Life. In Andrew Adamatzky, editor, *Collision-Based Computing*, pages 513–539. Springer London, 2002.

[31] J. M. Robson. N by N checkers is exptime complete. *SIAM Journal on Computing*, 13(2):252–267, 1984.

[32] J.M. Robson. The complexity of Go. *Proceedings of the International Federation for Information Processing*, pages 413–417, 1983.

[33] Ulf Rüegg, Steve Kieffer, Tim Dwyer, Kim Marriott, and Michael Wybrow. Stress-minimizing orthogonal layout of data flow diagrams with ports. In Christian Duncan and Antonios Symvonis, editors, *Graph Drawing*, volume 8871 of *Lecture Notes in Computer Science*, pages 319–330. Springer Berlin Heidelberg, 2014.

[34] Jeffrey Shen. Grid reductions. `https://github.com/jeffdshen/grid-reductions`, August 2016.

[35] Jeffrey Shen and Kevin Wu. Griduction - a framework for visualizing grid-based reductions. From 6.890 Algorithmic Lower Bounds: Fun with Hardness Proofs, a course at MIT, 2014.

[36] Roberto Tamassia. On embedding a graph in the grid with the minimum number of bends. *SIAM J. Comput.*, 16(3):421–444, 1987.

[37] Roberto Tamassia, Giuseppe Di Battista, and Carlo Batini. Automatic graph drawing and readability of diagrams. *IEEE Transactions on Systems, Man and Cybernetics*, 18(1):61 – 79, 1988.

[38] Giovanni Viglietta. Gaming is a hard job, but someone has to do it! In Evangelos Kranakis, Danny Krizanc, and Flaminia Luccio, editors, *Fun with Algorithms*, volume 7288, pages 357–367. Springer Berlin Heidelberg, 2012.

[39] Toby Walsh. Candy Crush is NP-hard. arXiv:1403.1911, March 2014.

[40] Takayuki Yato. On the NP-completeness of the Slither Link puzzle. *IPSJ SiG Notes*, AL-74:25–32, 2000.