

Automatic Protoboard Layout

by

Michael Mekonnen

B.S. EECS, Massachusetts Institute of Technology (2013)

B.S. Mathematics, Massachusetts Institute of Technology (2013)

Submitted to the Department of Electrical Engineering and Computer
Science

in partial fulfillment of the requirements for the degree of

Masters of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2014

© Massachusetts Institute of Technology 2014. All rights reserved.

Author
Department of Electrical Engineering and Computer Science
November 23, 2013

Certified by
Dennis M. Freeman
Professor
Thesis Supervisor

Certified by
Adam J. Hartz
Lecturer
Thesis Supervisor

Accepted by
Professor Albert R. Meyer
Chairman, Department Committee on Graduate Theses

Automatic Protoboard Layout

by

Michael Mekonnen

Submitted to the Department of Electrical Engineering and Computer Science
on November 23, 2013, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Electrical Engineering and Computer Science

Abstract

As an important component of the Circuits module of the first Introduction to Electrical Engineering and Computer Science course at MIT (6.01), students design and build several circuits over the course of three weeks. When working on the more intricate circuits, an unfortunately large proportion of students' lab time is spent on laying out the circuits on protoboards. This project introduces a new circuit schematic entry tool for 6.01 capable of automatically generating protoboard layouts for circuits that students may design in the course. The tool allows for students to easily build and analyze circuits through an intuitive graphical user interface, and generates layouts that are almost always aesthetically pleasing and easy to build and debug at the click of a button. The layout problem is solved by utilizing the *A** search algorithm exactly as presented in 6.01.

Thesis Supervisor: Dennis M. Freeman
Title: *Professor*

Thesis Supervisor: Adam J. Hartz
Title: *Lecturer*

Acknowledgments

First and foremost, I would like to thank Professor Freeman and Mr. Hartz for their nonstop support and encouragement from the very start of this project to its finish. They were always happy to talk with me, listen to my ideas, and give me constructive feedback whenever I asked for it. I am very happy to have worked with and developed a lasting relationship with great mentors.

Next, I would be remiss not to express my gratitude to my always loving and supportive family. I would not have been able to complete this project without the constant encouragement from my parents and siblings.

Finally, I would like to thank the Fall, 2013 members of the 6.01 staff, including Instructors, Teaching Assistants, and Lab Assistants, for testing the product and kindly giving me ideas on how to improve it.

Contents

1	Introduction	10
1.1	Problem Statement	10
1.2	Outline	11
2	Background	12
2.1	Technical Background	12
2.1.1	What are our circuit components?	12
2.1.2	What is a circuit schematic?	14
2.1.3	What is a protoboard?	14
2.1.4	What is a protoboard layout of a circuit schematic?	14
2.2	Previous Work	16
2.2.1	CMax	16
2.2.2	Current work in automatic layout	19
3	Methods	20
3.1	GUI	20
3.2	Solving the Layout Problem	20
3.2.1	Part 1: Placement	22
3.2.2	Part 2: Wiring	26
3.2.3	Combining the Methods	30
3.2.4	Evaluation	32
4	Results	36

4.1	Comparing Placement Methods	40
4.2	Comparing Wiring Methods	43
4.3	Comparing Search Methods	53
4.4	Combined Algorithm	58
4.5	Effect of limiting the number of vertices to expand in A^*	63
5	Discussion	69
5.1	Search Space Size	69
5.2	Justifying Placement Choices	70
5.3	Explaining the Results	71
5.3.1	Comparing Placement Methods	71
5.3.2	Comparing Wiring Methods	73
5.3.3	Comparing Search Methods	74
5.3.4	Putting Them All Together	75
5.4	Further Work	76
5.4.1	Treating Resistors as Wires	76
5.4.2	Building Layouts Similar to Previously Generated Layouts	77
5.5	Remarks	78
A	Schematic Drawing GUI	79
A.1	Palette	79
A.2	Board	81
A.3	Analysis	82
A.3.1	Simulation	82
A.3.2	Layout	82
A.4	Other Features	84
A.5	Shortcuts	84

List of Figures

2-1	6.01 robot.	13
2-2	Sample circuit schematic: head angular position controller.	14
2-3	Physical protoboard. In the top and bottom groups, the dots are interconnected horizontally. In the middle two groups, the dots are interconnected vertically.	15
2-4	Protoboard layout for the schematic in Figure 2-2.	16
2-5	CMax layout for the schematic in Figure 2-2.	17
3-1	Sample schematic drawn on the schematic entry tool.	21
3-2	Various acceptable ways of putting each of the circuit pieces on the protoboard.	23
3-3	Bases for random schematic generation.	33
3-4	Sample randomly generated schematic.	34
4-1	All possible alternatives to the algorithm.	36
4-2	Histogram of the complexity the 4425 schematics that we will use for evaluation.	38
4-3	Exemplar schematic.	39
4-4	Blocking vs. Distance.	40
4-5	Placement method comparison: success rates.	41
4-6	Placement method comparison: success rate trends.	42
4-7	Placement method comparison: wiring time trends.	43
4-8	Placement method comparison: layout quality trends.	44
4-9	Placement method comparison: layout badness trends.	45

4-10 All-pairs vs. Per-node, increasing vs. Per-node, decreasing vs. Per-pair, increasing vs. Per-pair, decreasing vs. Straight.	47
4-11 Wiring method comparison: success rates.	48
4-12 Wiring method comparison: success rate trends.	49
4-13 Wiring method comparison: wiring time trends.	50
4-14 Wiring method comparison: layout quality trends.	51
4-15 Wiring method comparison: layout badness trends.	52
4-16 A* vs. Best First Search.	53
4-17 Search method comparison: success rates.	54
4-18 Search method comparison: success rate trends.	55
4-19 Search method comparison: wiring time trends.	56
4-20 Search method comparison: layout quality trends.	57
4-21 Search method comparison: layout badness trends.	58
4-22 Combined algorithm exemplar.	59
4-23 Combined algorithm success summary.	60
4-24 Combined algorithm total CPU time trend as a function of circuit complexity.	61
4-25 Combined algorithm quality trend.	62
4-26 Combined algorithm layout badness trend.	63
4-27 Histogram of number of vertices expanded for all pairs wiring.	64
4-28 Histogram of maximum number of vertices expanded for per-node, increasing wiring.	65
4-29 Histogram of maximum number of vertices expanded for per-node, decreasing wiring.	66
4-30 Histogram of maximum number of vertices expanded for per-pair, increasing wiring.	67
4-31 Histogram of maximum number of vertices expanded for per-pair, decreasing wiring.	68
A-1 Parts of the schematic entry GUI.	80

A-2	Grouped components: Robot Connector and Head Connector.	80
A-3	Wire highlighting example.	83
A-4	Component highlighting example.	83

List of Tables

4.1	Placement method comparison: success rates.	41
4.2	Wiring method comparison: success rates.	49
4.3	Search method comparison: success rates.	54
5.1	Number of ways of packaging together n op-amps for various values of n	71
A.1	Shortcuts.	85

Chapter 1

Introduction

1.1 Problem Statement

In this paper we discuss the problem of automatic protoboard layout generation. Importantly, we are interested in automatically generating layouts that are aesthetically pleasing, easy to build, and easy to debug. The program that this paper discusses is strictly geared towards circuits that students would build in the Introduction to Electrical Engineering and Computer Science I[1] course at MIT (also known as 6.01).

Layout is generally a hard problem for humans. In 6.01, students are tasked with experimenting with various circuits. We are motivated to solve the layout problem so as to let students spend most of their lab time on thinking about how to design circuits and almost no time worrying about how to lay it out on a protoboard. This project aims to build a program that lets students very easily build and simulate circuit schematics through an intuitive graphical user interface (GUI). After building and testing their circuits with the program, students can then proceed to building the circuit on a physical protoboard based on the layout generated by the program. Not only does the program generate a layout, but it also makes it easy to relate the original schematic drawn by the student to the generated layout. With this program, a student's lab time will be spent mostly on designing, building, and testing circuits rather than on the much less instructive task of layout.

1.2 Outline

Chapter 2 describes in detail the terminology that will be used in this paper and explores the current infrastructure available for 6.01 students as well as previous work done in automatic layout generation. Chapter 3 discusses how we solve the problem, including various alternatives considered in each part of the solution, and how we evaluate our solution. Chapter 4 presents data to compare the various alternatives discussed in Chapter 3 and also evaluates the final algorithm on a large test dataset. Finally, Chapter 5 presents arguments for the choices made in our solution to the problem and also elaborates upon the results presented in Chapter 4.

Chapter 2

Background

In this section we will discuss essential background information to this project. First we discuss the specific terminology used in this paper. Next we discuss previous work done that relates to this project.

2.1 Technical Background

As already mentioned, this project aims to produce a new teaching tool for 6.01. As such, let us first discuss the scope of circuits in 6.01.

2.1.1 What are our circuit components?

The rudimentary circuit components that students work with are resistors, operational amplifiers (op-amps), and potentiometers (pots). Students start out by building very simple circuits, and then go on to building more complicated circuits with time. The simplest circuits that students build aim to control lego motors in a particular way. In constructing these circuits, students use 6-pin *motor connectors* to connect their circuits to lego motors. The more complicated circuits students build interact with robots that were built specifically for the purposes of 6.01. One of the 6.01 robots is displayed in Figure 2-1. Students use 8-pin *robot connectors* to connect their circuits to robots. The robots can be equipped with heads that have vision



Figure 2-1: 6.01 robot.

capabilities. Each head has a rod attached to a potentiometer. Also attached to the rod are a lego motor and a plate containing two photosensors positioned at a 90° angle from each other. The photosensors are used to serve as eyes for the robot. This setup allows us to turn the head by controlling the motor and inquire the current position of the head by probing the pot. Figure 2-1 displays a robot with a head. Students use 8-pin *head connectors* to connect their circuits to robot heads.

All together, our components are resistors, op-amps, pots, motor connectors, robot connectors, and head connectors.

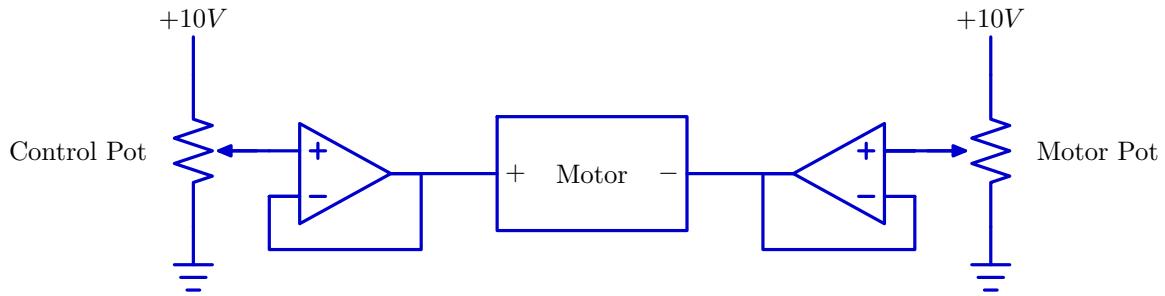


Figure 2-2: Sample circuit schematic: head angular position controller.

2.1.2 What is a circuit schematic?

Throughout this paper, the term *circuit schematic* will refer to a drawing or a sketch of a circuit containing its components and all the interconnections between the components drawn as wires. This is what one would sketch on a piece of paper in the process of designing a circuit. Figure 2-2 presents an example of a circuit schematic.

2.1.3 What is a protoboard?

Protoboards are constructs that make it easy to quickly build and test circuits. They present a 2-dimensional array of cleverly interconnected dots in which circuit pieces and wires can be inserted. Figure 2-3 presents an example of an empty physical protoboard. In the orientation depicted in Figure 2-3, a protoboard has 4 groups of rows: the first 2 rows, the next 5 rows, the next 5 rows, and finally the last 2 rows. In the first and last groups, the dots on the protoboard are interconnected horizontally. In the middle two groups, the dots on the protoboard are interconnected vertically. This interconnection scheme is depicted in Figure 2-3.

2.1.4 What is a protoboard layout of a circuit schematic?

The protoboard layout of a given schematic is the placement of circuit pieces and wires on a protoboard that corresponds to the schematic. This is done by placing the appropriate pieces on the protoboard and then appropriately interconnecting them with wires as prescribed by the schematic. As an example, Figure 2-4 presents the

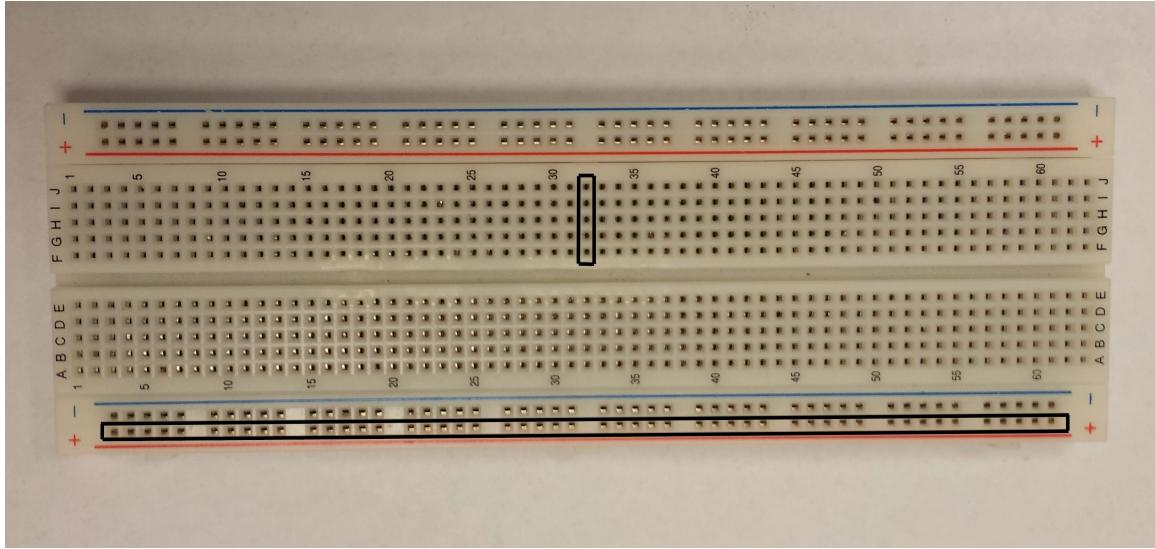


Figure 2-3: Physical protoboard. In the top and bottom groups, the dots are interconnected horizontally. In the middle two groups, the dots are interconnected vertically.

protoboard layout corresponding to the example schematic shown in Figure 2-2.

For each of the circuit components we are interested in, there is a corresponding circuit piece that may be inserted into the protoboard. The one exception is that op-amps come in pairs. That is, each op-amp circuit piece that is inserted in the protoboard actually contains two op-amp components within it. This raises an important design question when we layout a schematic: what is the best way to group together the op-amps in a schematic to result in the “best” layout? In answering this question, the designer must have some criteria for what makes a layout “good.” While there are no conclusive answers for this question, general rules of thumb are (in no particular order):

- The layout should not have any wires that cross circuit pieces.
- The layout should have no occlusions (i.e. crossing wires with identical orientations).
- The layout should have no crossing wires.

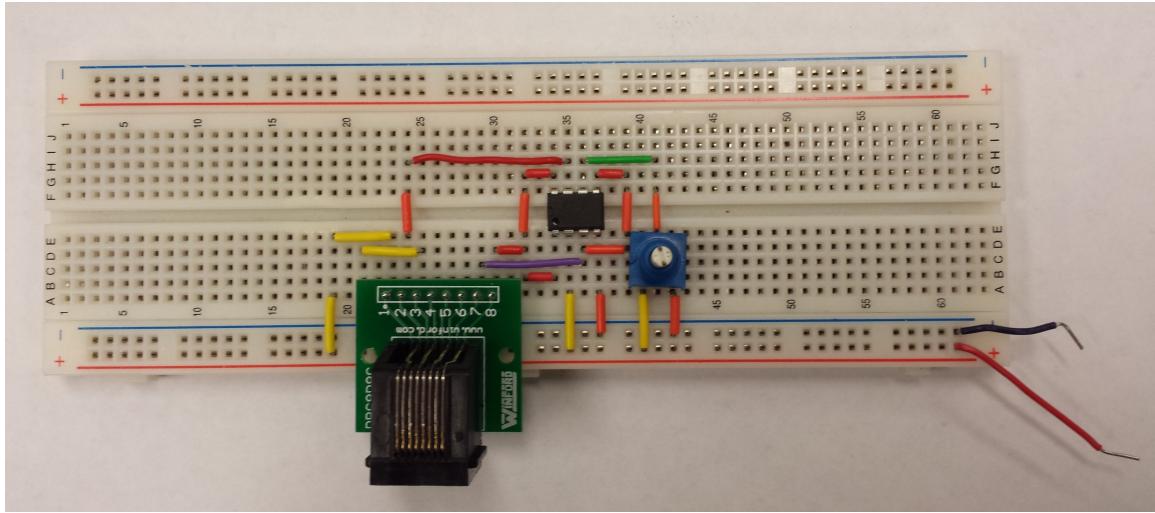


Figure 2-4: Protoboard layout for the schematic in Figure 2-2.

- The layout should only have horizontal and vertical wires (i.e. no diagonal wires).
- The layout should have as few wires as possible.
- The total length of wires in the layout should be as small as possible.

Given the background information discussed thus far, the goal of our project is generating a “good” protoboard layout from circuit schematics automatically.

2.2 Previous Work

Here we will discuss previous work that has been done relating to this project. First, as our project aims to augment the quality of 6.01, we look at the current infrastructure available for students. Next, we look at what work has been done relating to layout in general.

2.2.1 CMax

In a typical circuits lab in 6.01, students first design a circuit by drawing a schematic of the circuit on paper and discussing their design with a staff member. After they

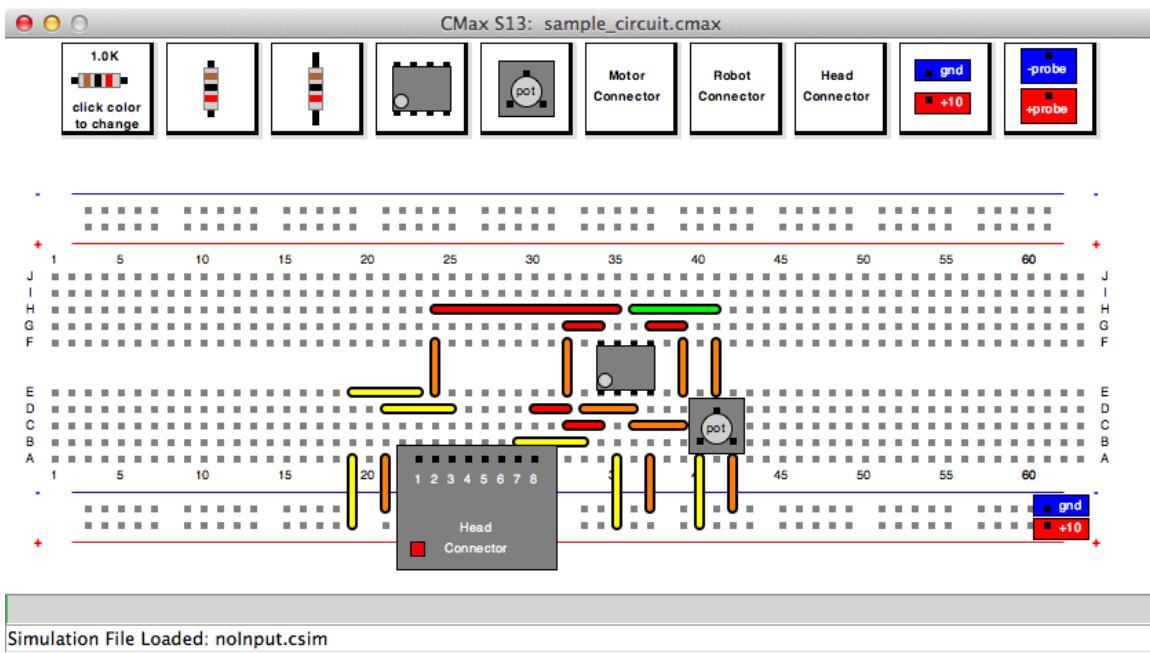


Figure 2-5: CMax layout for the schematic in Figure 2-2.

iteratively amend their design and are happy with it, they build the circuit on a simulation tool called Circuits Maximus (CMax)[2]. With this tool, students can layout their circuits on a simulated protoboard as if they were laying it out on a physical one. CMax allows students to simulate the circuit to make sure that it behaves as desired. Circuit layout is much easier on CMax than on a physical protoboard. Hence, CMax provides a very fast and safe way of debugging circuit layouts. Once the students are satisfied with their observations from CMax, they build their circuits on physical protoboards and carryout the appropriate experiments. Figure 2-5 presents a CMax layout of the schematic in Figure 2-2.

CMax has been a fantastic resource for 6.01 students. Its introduction has made learning circuits significantly easier for many students, especially those that have little or no prior experience with circuits. In addition to making the lab exercises much more manageable, it provides students with a very handy way to build, analyze, and experiment with circuits at their own leisure outside of lab.

While CMax is a fantastic tool, we can imagine a tool that can be even more useful. The most instructive part of the labs that students do in the circuits module of 6.01

is really designing the circuits in the first place, which they currently do by drawing schematic diagrams on paper. Once they are happy with their schematic diagrams, they proceed to laying out the corresponding circuits with CMax. The process of laying out a schematic does not really have very much instructive substance. This process is essentially solving a puzzle, and has almost nothing to do with the subject matter – designing circuits. In fact, when the circuits get complicated and involve many pieces, translating a schematic diagram into a protoboard layout gets to be quite challenging and time-consuming. In these situations, students often end up with convoluted and unpleasant layouts that are very difficult to debug in the likely case of the circuit not behaving as expected. Not only are such layouts difficult to debug for the students themselves, but they are also often difficult for staff members to fathom.

In the best case scenario, students should not have to produce protoboard layouts for their schematic diagrams. Indeed they should work out the right schematic diagram of the circuit of interest, but the layout generation should not be part of the learning process. This project aims to let students draw and analyze schematic drawings of circuits and produce the corresponding protoboard layouts automatically. Given the protoboard layouts output by this tool, students can proceed to building the circuits on physical protoboards and carrying out the appropriate experiments.

With this tool, a typical circuits lab would go as follows. First, as before, the students draw schematic diagrams of their circuits on paper. Once they have schematic drawings they are happy with, they can directly draw their schematic drawings on the simulation tool. In fact, students may go straight to building the schematic drawings on the simulation tool, bypassing the experimentation on paper. Once they have a schematic drawn, they can analyze it with the tool, discuss it with staff members, and amend it easily and quickly with the user-friendly GUI of the simulation tool. When they are satisfied with the behaviors of their schematic circuit, they can produce the corresponding protoboard layout simply at the click of a button – this would be the most important advantage of this tool. They can then build the layout on a physical protoboard and carryout experiments with it.

2.2.2 Current work in automatic layout

In my explorations, I was not able to find any tools that completely automatically convert circuit schematics into protoboard layouts. However, there do exists tools that perform partially- or fully-automatic Printed Circuit Board (PCB) layout. To my findings, most of these tools do not publish their algorithms and, rather, keep them proprietary. Hence, I was not able to build my work off of any existing products. In a sense, this project aims to build something new.

Chapter 3

Methods

In this Section, we will discuss our solution to the problem stated in Chapter 1 and various alternatives we considered along the way. First we will briefly introduce the schematic entry GUI. Next, we will discuss in detail how we solved the protoboard layout problem and how we evaluated our solution.

3.1 GUI

We designed the schematic entry GUI to have a rich set of features so as to make drawing schematics a very easy and intuitive task for students. Figure 3-1 gives a version of the schematic drawn in Figure 2-2 as drawn of the schematic entry tool. Appendix A discusses the features and capabilities of the schematic entry GUI in much further detail.

3.2 Solving the Layout Problem

In broad terms, we solved the layout problem by formulating it as a search problem. By this I mean, given a schematic of a circuit, we start from an empty protoboard, and search through the space of all possible protoboard layouts to find the protoboard corresponding to the schematic at hand. Importantly, we utilize various simplifications and heuristics to facilitate the search as the space of all possible protoboards is

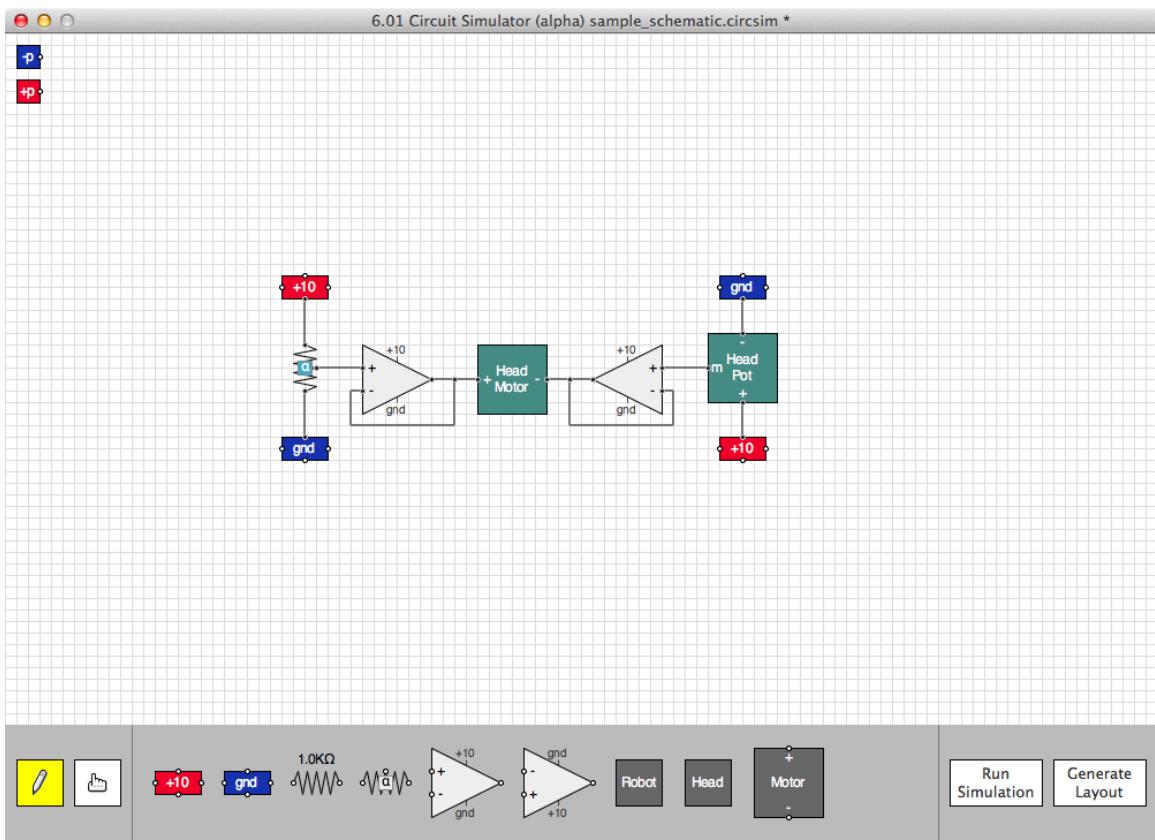


Figure 3-1: Sample schematic drawn on the schematic entry tool.

very large.

We broke down the problem into two parts. The first task is finding a placement of all the circuit pieces on the protoboard. The second task is wiring them up appropriately.

3.2.1 Part 1: Placement

Let us first consider how to place a set of circuit pieces on the protoboard for a given circuit schematic. Any given circuit may contain resistors, op-amps, pots, motors, head connector parts, or robot connector parts. For each of these components, we must put down a corresponding piece on the protoboard. As each piece may be placed on the protoboard in one of many different ways, we first decided on a fixed set of allowed placements for each of the pieces. Figure 3-2 presents these acceptable placements. Resistors are placed in the middle strip of the protoboard. Op-amp pieces are also placed in the middle strip of the protoboard, but with two possible orientations. Op-amp pieces are unique in that each op-amp pieces contains two op-amps within it. Thus, we face the task of packaging the op-amps in the schematic in the “best” possible way, i.e. so as to require as little work as possible when wiring the pieces together. Section 5.2 more precisely discusses the number of possibilities. Pots have two possible vertical positions as well as two possible orientations. The connector pieces have two possible vertical positions each.

When choosing a placement of circuit pieces on the protoboard, we have at hand a plethora of options. First we must choose among a possibly large number of ways to package together the op-amps in the circuit. For each possible packaging of op-amps, we must consider various ways of placing the pieces on the protoboard, even with the restrictions put forth above.

Simplifications

We reduce this large number of possibilities by only allowing placements in which no two pieces share a column. This is not necessary in general, but the number of pieces

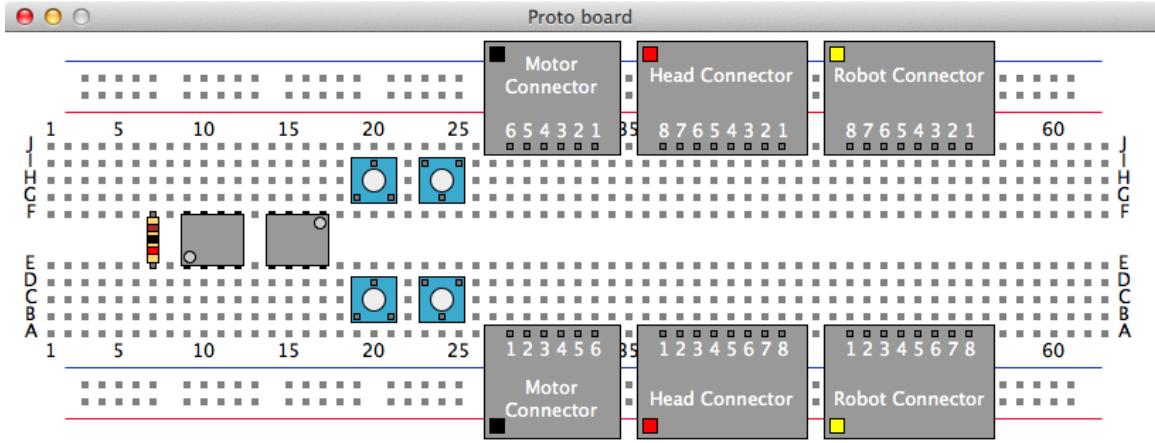


Figure 3-2: Various acceptable ways of putting each of the circuit pieces on the protoboard.

necessary for a typical 6.01 circuit would certainly fit in this framework.

Next, we specify that there be exactly two columns on the protoboard separating each consecutive pair of pieces, unless the pieces are both resistors, in which case there must be exactly one column separating them. These numbers of columns were chosen to leave enough space for wiring. Given a set of pieces to be put on the protoboard, this specification reduces the problem of choosing a placement for the pieces to finding an *order* of the pieces together with choosing their respective vertical locations and orientations.

Given these simplifications, we have various options as to how to pick a placement.

Random Placement

One simple alternative may be to choose a placement randomly. That is, we choose an op-amp packaging randomly; we choose an order of the pieces randomly; and we choose the vertical locations and orientations of the pieces randomly as well. The advantage of this approach is that it gives us a placement very quickly without requiring much computation. On the other hand, we may end up placing two pieces that need to be connected to each other very far apart, and we will have a difficult time doing the wiring. Hence, we ought to consider alternatives in which we take into

account the task of wiring. We should try to place the pieces so as to require as little work during wiring as possible.

Minimal Heuristic Cost

The key idea is that if two pieces are meant to be connected together by wires, then they ought to be placed close to each other on the protoboard. We can capture this idea by assigning heuristic costs to the placements and choosing a placement that produces the minimal heuristic cost. To that end, there are two heuristic cost functions we considered.

Distance Based Cost Given a circuit schematic and a corresponding placement of the circuit pieces on the protoboard, what do we need to connect with wires? Well, every pair of components in the schematic that are connected by a wire gives us a corresponding pair of locations on the protoboard that ought to be connected by wires. However, we can express this requirement a little bit more concisely. We ought to consider all of the nodes in the schematic, and find the circuit components in the schematic that are connected to the respective nodes. Now for each node in the circuit, we get a set of locations on the protoboard that ought to be interconnected. The first step in devising the distance based cost function is to have a way to estimate the cost of connecting two locations on the protoboard. A simple such cost function that comes to mind is the Manhattan distance between the two locations. Recall that we want to produce layouts that only contain horizontal and vertical wires (i.e. no diagonal wires) so the Manhattan distance cost is appropriate. Given this heuristic cost for connecting two locations with wires, we can define the heuristic cost for interconnecting the locations associated with a particular node to be the weight of the minimum spanning tree of the locations. Now we can define the cost of a placement to be the sum over all nodes in the circuit of the cost for interconnecting the locations for each node.

Blocking Based Cost The most scare resource on the protoboard are the rows. We are given 10 rows with which to work, and in producing a layout, we must fit all the wires within these 10 rows. This gives us an idea on how to define how hard a placement will be to wire. Given a set of pieces, we can find a set of pairs of locations on the board that need to be connected as we did above. As noted in Section 2.1.3 there are two groups of 5 rows on a protoboard. Now for each group of rows and each column in the group of rows, we can count how many pieces reside in that column, and how many wires may pass through the column (in connecting a given pair of locations assuming an empty protoboard). Our goal will be to minimize the number of columns that would be used heavily. In our implementation, the final cost is computed as the sum of the squares of the counts for each column.

Using one of the two cost functions discussed above, we can aim to find a placement with the minimal cost. However, this involves trying all possible orderings of the pieces with which we are working. For example, if we are trying to order 10 pieces, we would need to look at $10! = 3628800$ possible orderings. Note that this is in addition to searching over all possible ways of packaging the op-amps together. It is clear to see that the search for a minimal cost placement quickly gets out of hand. So we aim to find a placement that has a very small, though maybe not minimal, cost.

Small Heuristic Cost

Algorithm 1 presents a polynomial-time procedure that orders a given list of pieces in a way that results in a small cost. The algorithm places one of the pieces at a time, starting from an empty placement. It relies on two ideas. First, once a piece has been placed, all the pieces that are connected to it will be placed soon after so that it is more likely that those pieces are placed close to it. Second, we place the pieces with the most nodes first since those are the ones that most likely have connections with many other pieces.

Using one of the above methods, we can find a placement of circuit pieces on a protoboard. Our next task is wiring them together to produce a circuit equivalent to the circuit schematic of interest.

Algorithm 1: Producing a circuit piece placement with small heuristic cost.

Data: A list P of circuit pieces.

Result: A list R of circuit pieces representing a placement.

Sort P in decreasing number of nodes on the respective pieces

$Q \leftarrow$ empty Queue

$R \leftarrow$ empty List

while P is not empty **do**

Pop the first piece in P and push it onto Q

while Q is not empty **do**

$p \leftarrow Q.pop()$

Consider all vertical locations and orientations of p

Place p at an index in R that minimizes the cost of P

foreach piece q in P connected to p **do**

Pop q out of P and push it onto Q

3.2.2 Part 2: Wiring

In the previous section we discussed what locations we need to wire together: for every node in the circuit, we get a set of locations on the protoboard that need to be interconnected. The question now is how to achieve this wiring. We approach the problem as a search problem and use the A^* search algorithm to solve it. In fact the wiring step uses an infrastructure for the A^* search algorithm exactly as presented in the Search module of 6.01. Hence students in the class may very much appreciate an application of something they learned earlier in the course to produce a tool that they are using for something that may seem completely unrelated.

Using A^*

The A^* algorithm can be used to search for a path from some starting vertex ¹ in a graph to some goal vertex ². The algorithm works by keeping track of an agenda of vertices to consider in a priority queue where the value associated with each vertex in

¹The preferred terminology is “a node in a graph” but here we will use the term “vertex” since we already use “node” to refer to nodes in circuits.

²In fact, A^* guarantees an optimal path, a path that has the minimum possible cost from the starting vertex to a goal vertex, if we use a heuristic that is *admissible*. A heuristic is said to be admissible if it does not overestimate the actual minimal cost to a goal vertex for any state. Here, however, we will not worry about the admissibility of our heuristic as our main goal is designing the heuristic to prune as many states as possible.

the priority queue is the sum of the cost to get from the start vertex to the vertex at hand and the value of the heuristic computed at the state of the vertex. At each step of the algorithm, we pop one vertex from the priority queue (the vertex with the minimum associated value). If the vertex happens to satisfy the goal of the search, we return the state for that vertex as the answer to the search problem. Otherwise, we add the children vertices of that vertex to the priority queue and continue. When adding children vertices, we take care not to reconsider states that we have already considered via a different path. We call the process of popping a vertex from the priority queue and treating it as described *expanding* the vertex. In general, when using the *A** algorithm, we need to design four things:

1. The notion of a vertex in the search tree, the cost associated with a vertex, and how we obtain the neighbors of a vertex,
2. The starting vertex,
3. How we identify whether a particular vertex in the search tree achieves the goal of the search, and
4. A heuristic function that estimates the distance from a given vertex to a goal vertex.

Vertices

Each vertex will hold a representation of some protoboard. Each representation will contain all of the pieces, and possibly a set of wires interconnecting the pieces. The starting vertex will have all of the pieces but no wires.

We obtain the neighbors of a vertex by taking the current protoboard and producing new ones in which we place exactly one wire at various locations. We choose the starting point of a wire to be any one of the free locations on the protoboard that is already connected to one of the pieces, and we extend the wires in all possible vertical and horizontal directions up to some fixed wire length. Note that we need to

take great care when placing wires in order not to short different nodes. We discard any vertices that arise from placing a wire that shorts two different nodes.

The way we define the cost of a vertex, i.e. the cost of getting from the starting vertex to a vertex of interest, depends on our definition of a good protoboard layout. In general, we want to penalize having long wires, many wires, or crossing wires. In our implementation, while we have a large penalty for two crossing wires of opposite orientations (i.e. vertical and horizontal), we do not allow occluding (i.e. crossing wires that have the same orientation) as this configuration is particularly difficult to physically build and debug. Finally, we want to favor making a desired interconnection between locations on the protoboard.

Each vertex will not only hold a protoboard, but it will also hold a set of pairs of locations on the protoboard that need to be connected by wires. Each pair (loc_1, loc_2) of locations tells us that we need to have a set of wires connecting some location connected to loc_1 to some location connected to loc_2 .

An important consideration we need to make is how we want to organize the search. Recall that we have a set of nodes in the circuit of interest, and for each node we have a set of locations that need to be interconnected. Given this information, we may choose one of the following three strategies to carryout the search:

1. All pairs: For each node, collect a set of pairs of locations on the protoboard corresponding to a minimum spanning tree of the locations for that node, so that if all pairs of locations in this spanning tree are connected, then the locations for the node will be interconnected. Collect all such pairs of locations for all of the nodes in the circuit, and have the starting vertex hold this set of pairs of locations.
2. Per-node: Treat each node separately. That is, iteratively interconnect the locations for each of the nodes until there are no more nodes in the circuit.
3. Per-pair: Treat each pair of locations that needs to be connected separately. That is, iteratively connect pairs of locations that need to be connected until there are no more pairs.

The choice of one of these strategies has a significant effect on the outcome of the search. We will look at the difference in detail in Chapter 4.

Goal test

Given a vertex, we know that it is a goal vertex if all of the pairs of locations it holds are already connected in the protoboard it holds.

Search heuristic

In A^* search, choosing the right heuristic can often make the search much more efficient. Given a vertex, we can estimate its distance from a goal as follows. For each pair of locations (loc_1, loc_2) that need to be connected, we could consider its distance from the goal to be the smallest Manhattan distance between any location connected to loc_1 and any other location connected to loc_2 . To compute the heuristic cost of a vertex, we simply add up this value for each of the pairs of locations that need to be connected. In Chapter 4 we will compare the performance of A^* with this heuristic versus carrying out Best First Search with this heuristic. In Best First Search, as opposed to in A^* , we consider the cost of a vertex to be the value of the heuristic function at that vertex, and nothing more.

Limiting the number of expanded vertices

With the implementation of A^* we have discussed so far, the algorithm terminates if we either find a solution, or we exhaust the search space without finding a solution. In our search problem, the number of states is very large, so this implementation of A^* may sometimes run out of memory during the search. To mitigate this problem, we introduce a limit to the number of vertices the algorithm expands before giving up. That is, if the algorithm expands a certain fixed number of vertices and still has not found an answer, the algorithm gives up. For us, this limit is set to 300 vertices. In Chapter 4 we provide data to justify this choice.

3.2.3 Combining the Methods

With the methods discussed so far, we aimed to completely solve the layout problem with one placement method and one wiring method. However, as we will soon see, such an algorithm is bound to fail on some set of schematics. When we ultimately put the final algorithm in front of students, we would like to avoid failure. The algorithm should always be able to generate layouts. Generating a layout with a few diagonal or crossing wires is better than silently failing and leaving the student empty handed. Here, we will discuss how we combine the methods described so far into one layout algorithm. The motivation for this combination will be discussed in Chapter 5 based on the data we obtain for the alternatives described above. Algorithm 2 presents the combination algorithm.

Algorithm 2: Layout algorithm obtained by combining multiple ideas.

Data: A circuit schematic C .

Result: A layout corresponding to C .

```
foreach Placement cost metric  $M$  in (DISTANCE, BLOCKING) do
     $P \leftarrow$  Placement for  $C$  by using cost metric  $M$ .
    Connect the top and bottom rails on  $P$ .
    foreach Order  $O$  in (INCREASING, DECREASING) do
        pairs  $\leftarrow$  Pairs of location on  $P$  to connect given schematic  $C$  and
        connection order  $O$ .
        foreach  $(loc_1, loc_2)$  in pairs do
            Attempt to connect  $loc_1$  and  $loc_2$  on  $P$ .
            If successful, update  $P$  accordingly and then post-process  $P$ .
            If not, record that the pair  $(loc_1, loc_2)$  was not successfully
            connected.
        If all pairs are successfully connected, return  $P$ .
    Pick "best" unfinished layout.
    Connect remaining pairs with shortest possible wires (possibly diagonal).
    Post-process and return resulting layout.
```

There are a few key items in Algorithm 2. The algorithm works by attempting to solve the problem in four different ways: two different ways of doing placement together with two different orders of wiring pairs. Note, therefore, that we have chosen per-pair wiring. If any one of the four trials succeeds, the algorithm immediately returns the corresponding layout. If all four trials fail, on the other hand, the algo-

rithm picks the “best” partial solution and completes that solution by placing possibly diagonal wires, “best” being the partial solution that would require the least amount of additional wiring. This last step makes it highly unlikely that the algorithm will ever fail. The only way for the algorithm to fail is for there to be two nodes that need to be connected where all of the protoboard locations for at least one of the nodes is occupied, which is highly unlikely to happen. This high success rate comes at the cost of placing wires that will almost surely cross other wires on the board.

The algorithm automatically starts out by connecting the top and bottom rails of the protoboard so that all rail rows are used to connect to power and ground, and no other nodes. This restriction was not enforced before, but it is a restriction that certainly makes debugging easier.

The algorithm also has a post-processing step that attempts to improve the layout. The post-processing step makes three types of simple changes to the layout. First, we throw away any wires that do not serve to connect two parts of the circuit. That is, we throw away wires which have one end connected to the circuit, and the other end connected to nothing at all. This could happen in the search done by the wiring step, though very rarely. Second, we truncate long vertical wires into an equivalent set of smaller wires. For example, a wire going from one of the top rails to one of the bottom rails can be replaced by three smaller wires making the same connection. This change frees up rows for subsequent connections. Finally, if shifting a horizontal wire up or down results in fewer wire crosses, we make that change.

The last important aspect of this final algorithm not explicitly stated in Algorithm 2 is that, as it is the algorithm that will be put in front of students, it will be allowed to use wires of a select few lengths. The kits that students work with certainly do not come with wires of all lengths, so we force the wiring step to use wires of only those allowed lengths. We also avoid using length-1 wires as they are very difficult to insert and remove from protoboards.

3.2.4 Evaluation

Here we present how we go about evaluating a particular solution to the problem. How can we tell if a layout tool is good? In particular, how can we tell if a layout tool is good enough for the purposes of 6.01 labs. To answer these questions well, we need to test the layout tool on numerous schematics and analyze its performance on laying out those schematics. As manually generating numerous test schematics is tedious and very time-consuming, we devised a method to randomly generate thousands of test schematics.

The random schematic generation goes as follows. We create 6 basic sub-parts of schematics. These 6 bases are depicted in Figure 3-3. These bases cover all of the components that may be necessary in a 6.01 circuit. Each sub-part also offers at least 3 points of connection with other subparts. The random generation algorithm takes all possible combinations of 6 bases, allowing for repetition of bases with some restrictions. The Head Connector and Robot Connector bases can appear at most once as there is no need for more than one of each of these in 6.01 labs. The pot-follower base (i.e. the base that contains one pot and a follower op-amp) can appear at most twice as we never need more than two pots in 6.01 circuits. The motor base can also appear at most twice as we never need more than two motors per circuit in 6.01 labs. The other bases, T-resistor configuration and voltage divider, can be repeated up to 6 times. For a given combination of bases, we generate up n schematics, where n depends on the number of bases in the configurations (larger for configurations with more bases in them). We generate the n schematics by choosing a number of interconnections between the bases from 0 to $n - 1$. For each number of interconnections, we generate a schematic in which we put that many randomly chosen wires interconnecting the bases in the combination. Figure 3-4 presents a sample randomly generated schematic.

This scheme produces a total of 4425 test schematics out of a possible total of approximately $1.2e27$. When testing a particular algorithm on these test schematics, we run the algorithm on each test schematic 10 times. Chapter 4 presents the data

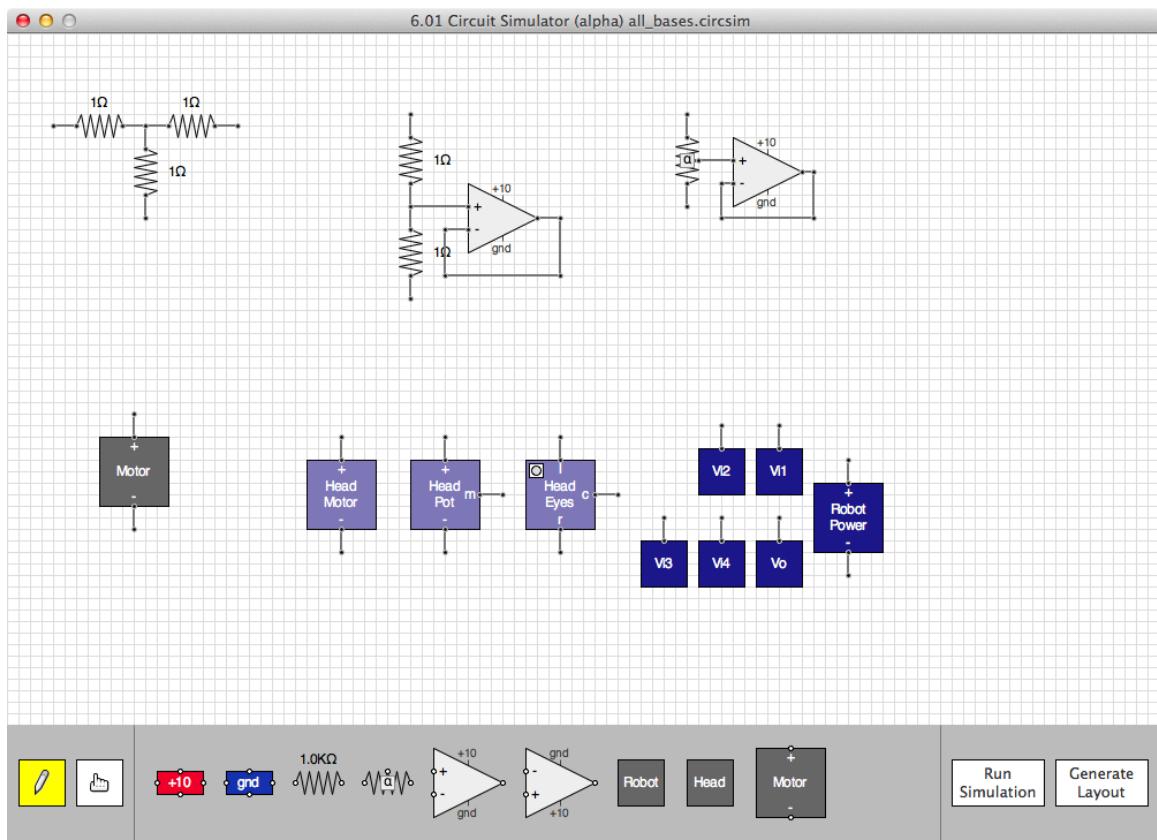


Figure 3-3: Bases for random schematic generation.

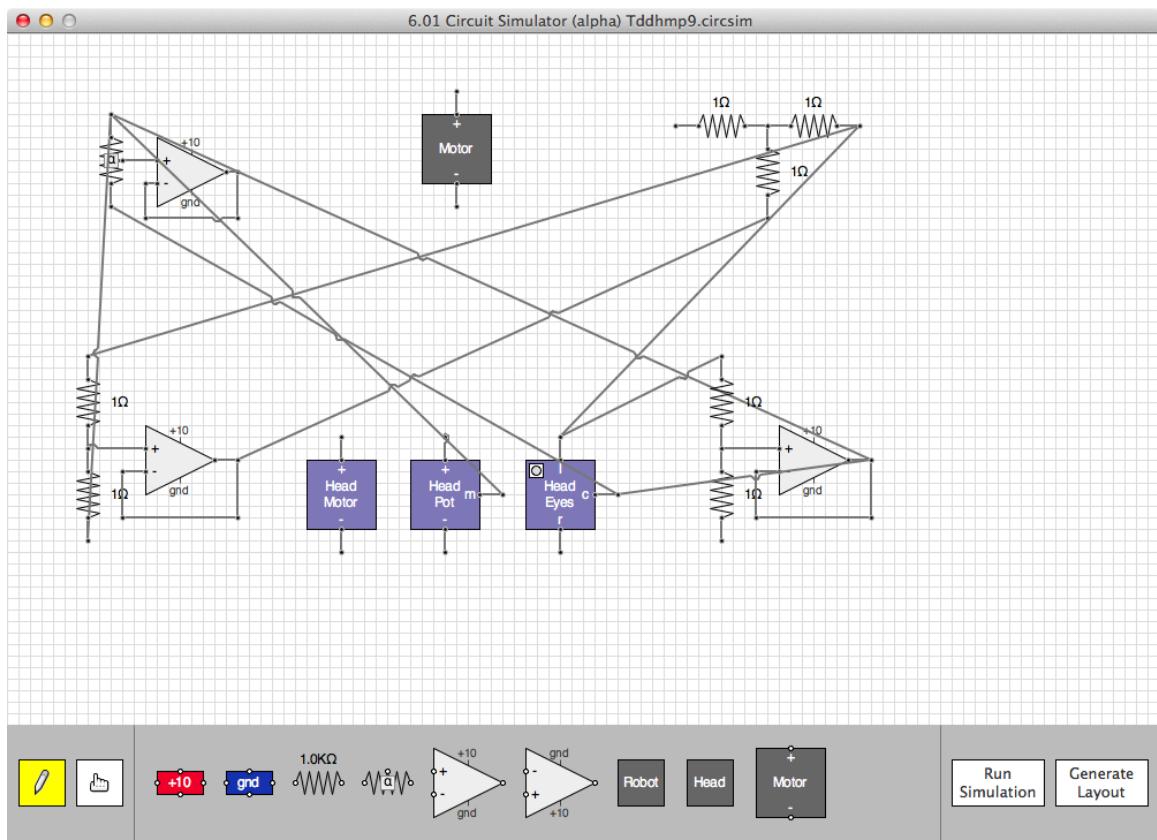


Figure 3-4: Sample randomly generated schematic.

collected in this manner comparing the various alternatives discussed in this Chapter.

An important question we must answer is how we quantify the goodness (or badness) of a particular layout. Our approach takes a weighted sum of a particular set of attributes of a given layout, where the weights are chosen comparatively. We set the badness of a layout to be:

$$\begin{aligned} & 1 \cdot \text{NUMBER_OF_WIRES} + \\ & 2 \cdot \text{TOTAL_WIRE_LENGTH} + \\ & 10 \cdot \text{NUMBER_OF_WIRE_CROSSES} + \\ & 10 \cdot \text{NUMBER_OF_DIAGONAL_WIRES} + \\ & 50 \cdot \text{NUMBER_OF_WIRE-PIECE_CROSSINGS} + \\ & 500 \cdot \text{NUMBER_OF_WIRE_OCCLUSIONS}. \end{aligned}$$

We will use this metric to decide which of a given set of alternatives tends to produce the better layouts. The weights in the metric were chosen to reflect how bad each of the attributes are relative to one another. This choice of weights, therefore, reflects the following reasonable set of statements. Recall that our goal is to produce layouts that are easy to build, easy to debug, and aesthetically pleasing.

- Having an additional wire is about as bad as increasing the total wire length on the protoboard by 2.
- Having two wires that cross is about as bad as increasing the total wire length on the protoboard by 5.
- A diagonal wire is about as bad as two wires that cross.
- Having 10 wires that cross is about as bad as having a wire that crosses a circuit piece.
- Having 10 wires that cross circuit pieces is about as bad as having a wire occlusion. The penalty on wire occlusions is meant to indicate that we want absolutely no wire occlusions in our layouts.

Chapter 4

Results

In Chapter 3 we discussed a general solution to the protoboard layout problem, and various alternatives that can be used in implementing the solution. Figure 4 presents the alternatives in a structured way. Here, we will explore these alternatives and compare them quantitatively. This section will provide data that is useful in comparing the alternatives, and the data will be discussed in detail in Chapter 5.

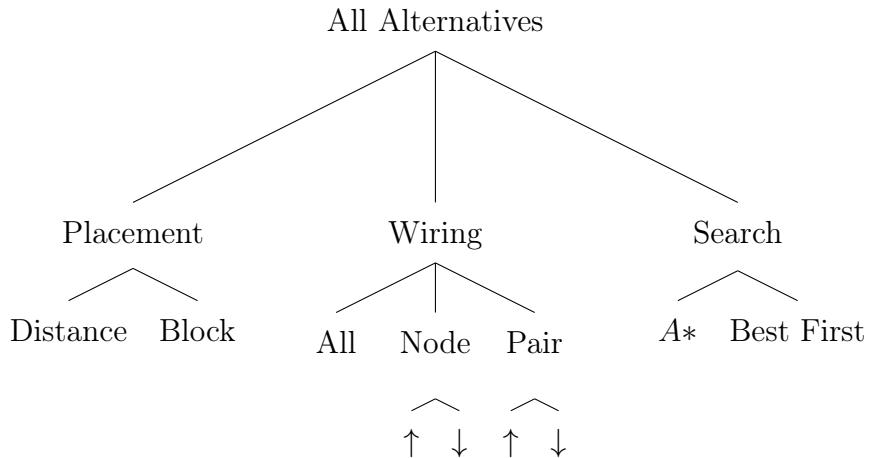


Figure 4-1: All possible alternatives to the algorithm.

As comparing all 20 possible implementations of the algorithm is tedious, we will compare the alternatives for each aspect of the algorithm while holding other aspects fixed. Hence, we will carryout the following comparisons:

1. Placement: Distance vs. Blocking. Wiring method will be per-pair, decreasing,

and we will use A^* .

2. Wiring: All pairs vs. Per-node, decreasing vs. Per-node, increasing vs. Per-pair, decreasing vs. Per-pair, increasing. Placement method will be blocking, and we will use A^* .
3. Search: A^* vs. Best First. Placement method will be blocking, and wiring method will be per-pair, decreasing.

The data to compare the alternatives is gathered as described in Chapter 3. We run the algorithm on 4425 randomly generated schematics of varying complexities. The algorithm is run 10 times on each schematic.

In comparing alternatives, there are 3 items we will consider:

1. Which alternative is the most successful?
2. Which alternative, when successful, takes the least amount of time?
3. Which alternative, when successful, produces the best layouts?

In comparing success, we will look at bar graphs of the number of successes on each of the 4425 schematic out of the 10 runs. We will also look at tables that provide the same data in more detail. To get an understanding of how the success rates vary with complexity, we will look at plots of circuit complexity versus success rate, where our measure of circuit complexity will be the number of pins in the circuit. A pin in a circuit is a connection point on a circuit component that is connected by wires to another connection point (on the same component or a different component). The number of pins in the circuit is a reasonable metric for how complex the circuit is. Figure 4-2 presents a histogram of the number of pins in the schematics in dataset that will be used to do all comparisons in this Chapter.

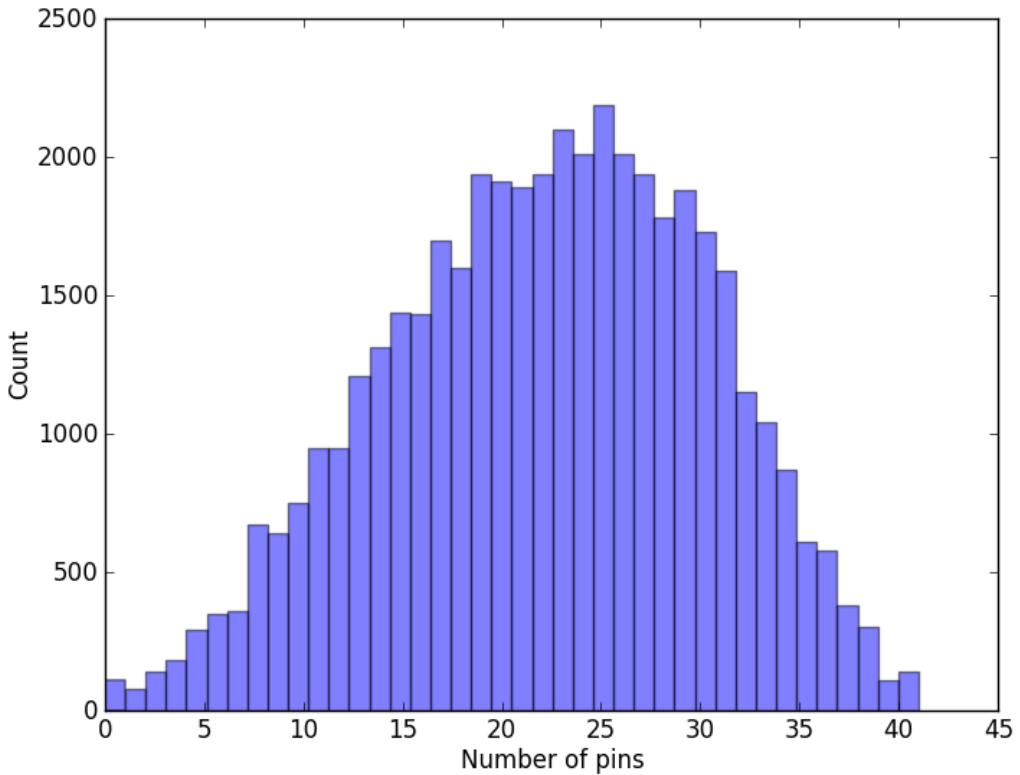


Figure 4-2: Histogram of the complexity the 4425 schematics that we will use for evaluation.

In comparing success time, we will look at CPU time spent on the wiring step, as the placement step has much less variability. We will look at plots of circuit complexity versus wiring times to do the comparisons.

In comparing the goodness of layouts, we will compare numbers of wires, total lengths of wires, numbers of wire crosses, and our layout badness metric as functions of circuit complexity.

Below we will present the data for each of the comparisons outlined above. Note that in all figures that follow, error bars indicate 1.96 times the standard error.

For each comparison, we present exemplar layouts generated by the alternative methods for the schematic shown in Figure 4-3.

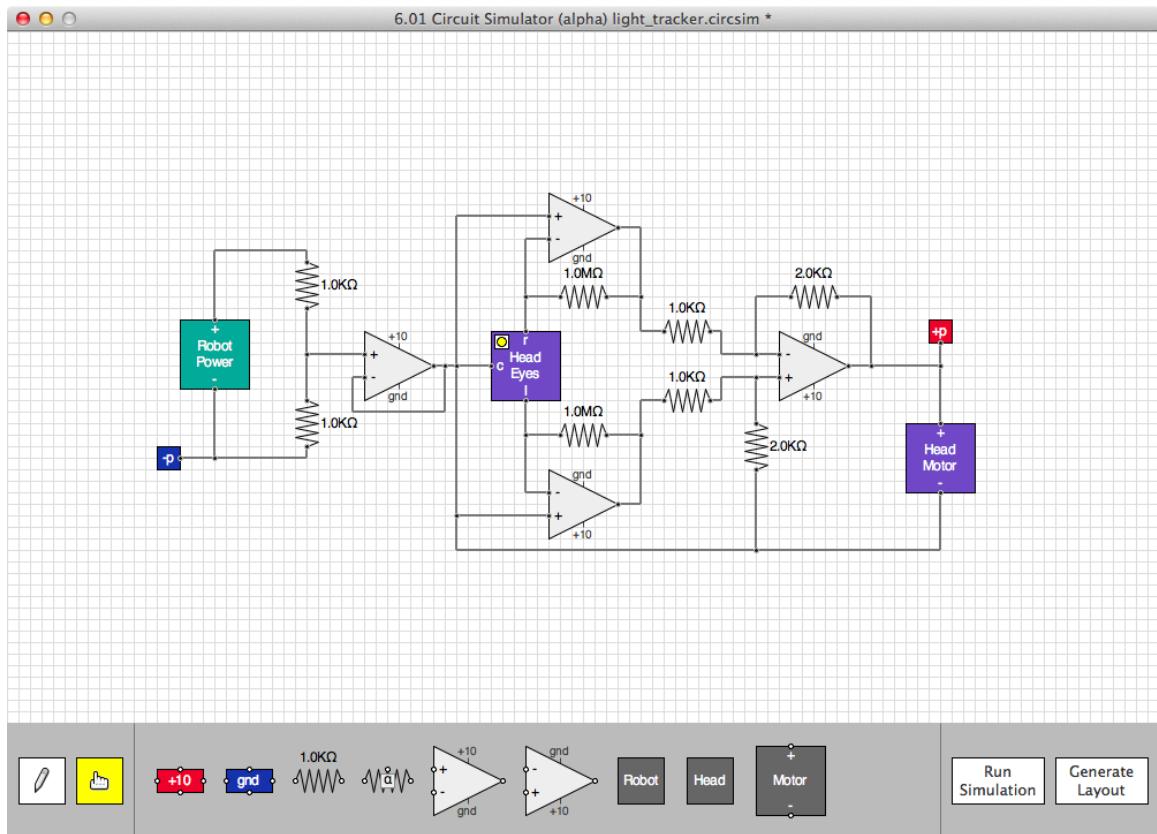


Figure 4-3: Exemplar schematic.

4.1 Comparing Placement Methods

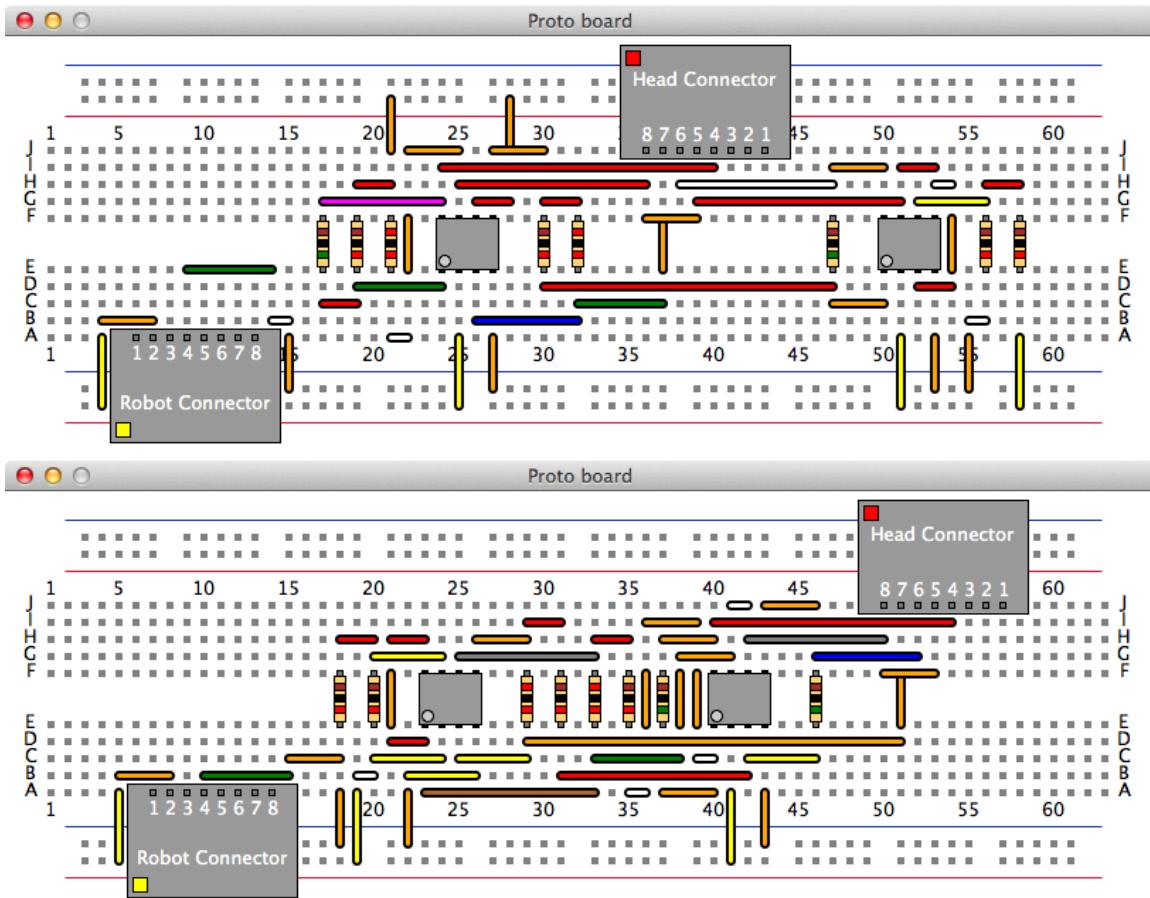


Figure 4-4: Blocking vs. Distance.

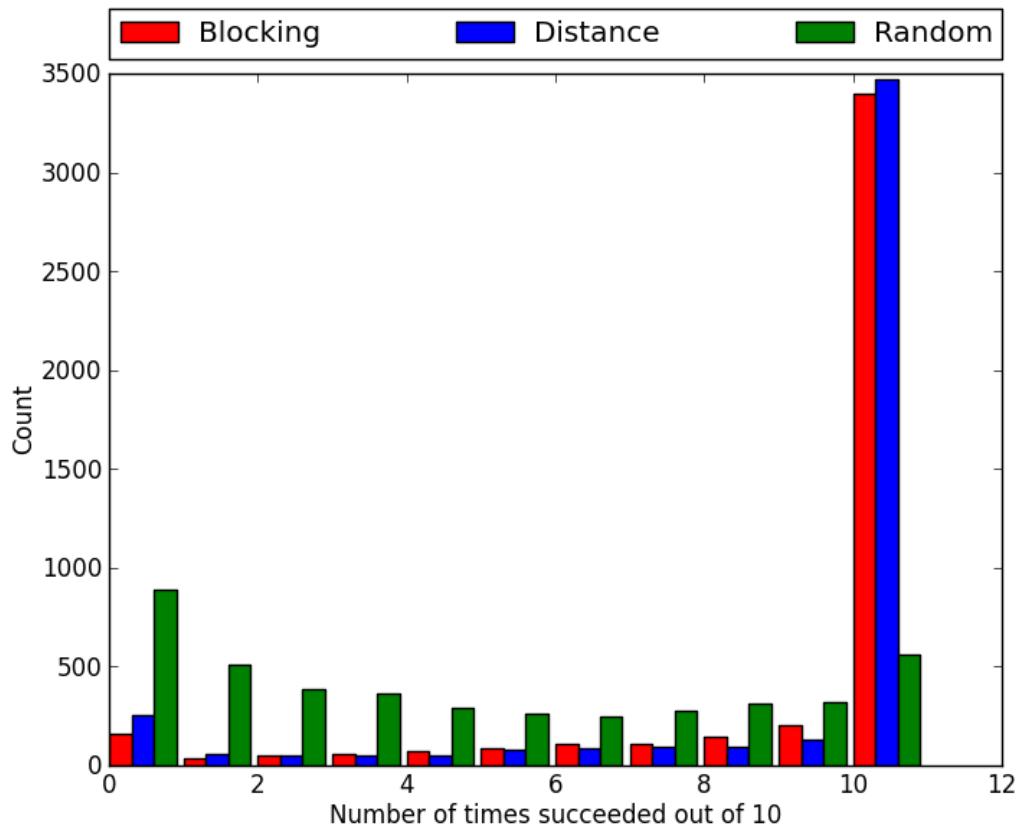


Figure 4-5: Placement method comparison: success rates.

	Number of times succeeded out of 10										
	0	1	2	3	4	5	6	7	8	9	10
Blocking	162 0.04	38 0.01	51 0.01	57 0.01	72 0.02	85 0.02	109 0.02	106 0.02	144 0.03	203 0.05	3398 0.77
Distance	258 0.06	55 0.01	54 0.01	50 0.01	52 0.01	77 0.02	86 0.02	97 0.02	93 0.02	130 0.03	3473 0.78
Random	893 0.20	512 0.12	387 0.09	364 0.08	292 0.07	259 0.06	247 0.06	277 0.06	311 0.07	321 0.07	562 0.13

Table 4.1: Placement method comparison: success rates.

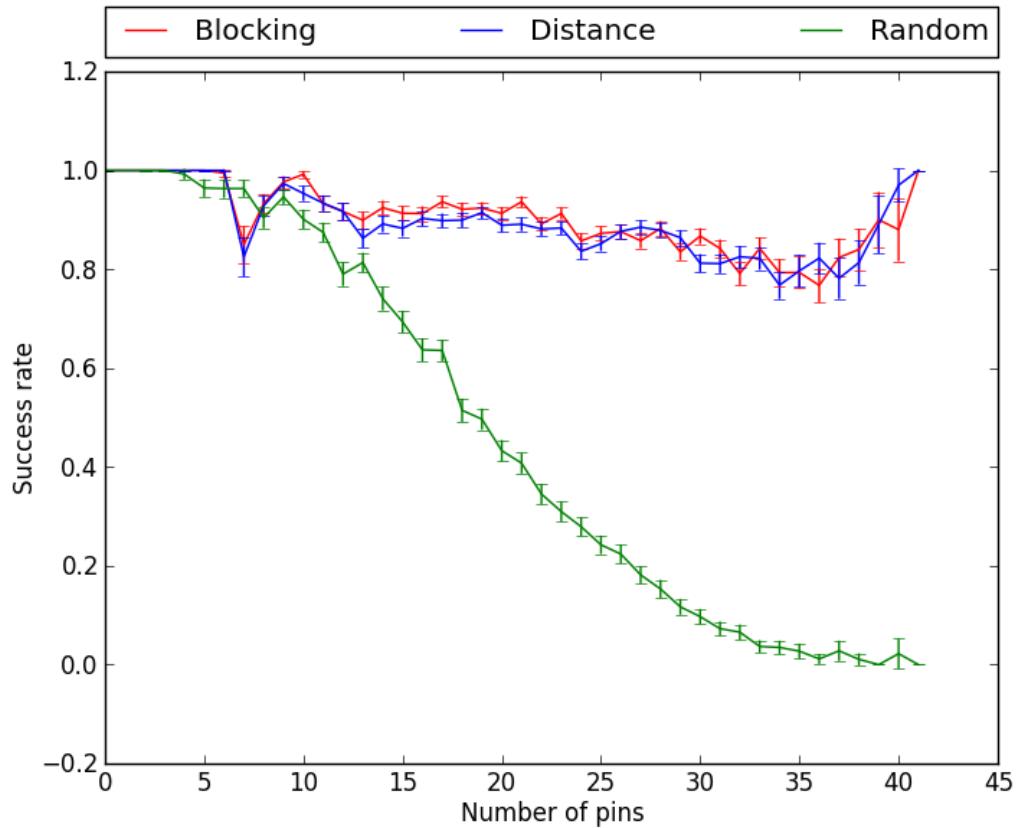


Figure 4-6: Placement method comparison: success rate trends.

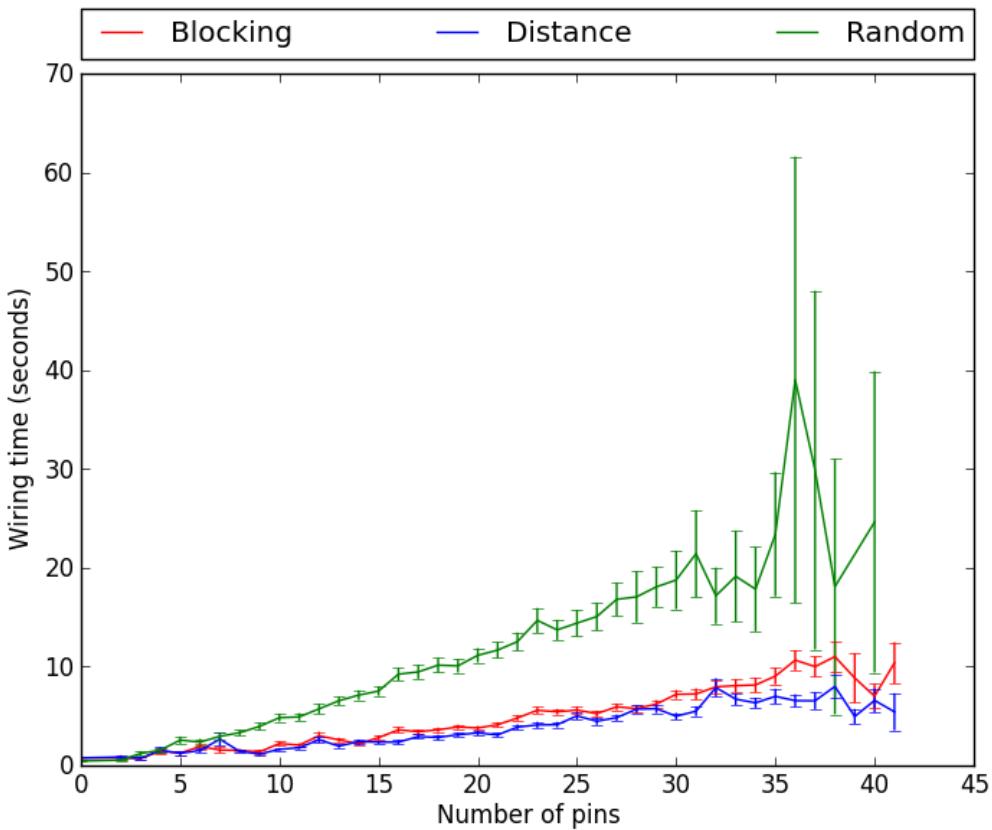


Figure 4-7: Placement method comparison: wiring time trends.

4.2 Comparing Wiring Methods

We used the distance based placement method instead of the blocking based placement method to generate the per-node, decreasing exemplar as the the combination of the blocking based method with this wiring method consistently failed on the schematic we are using to generate exemplars.

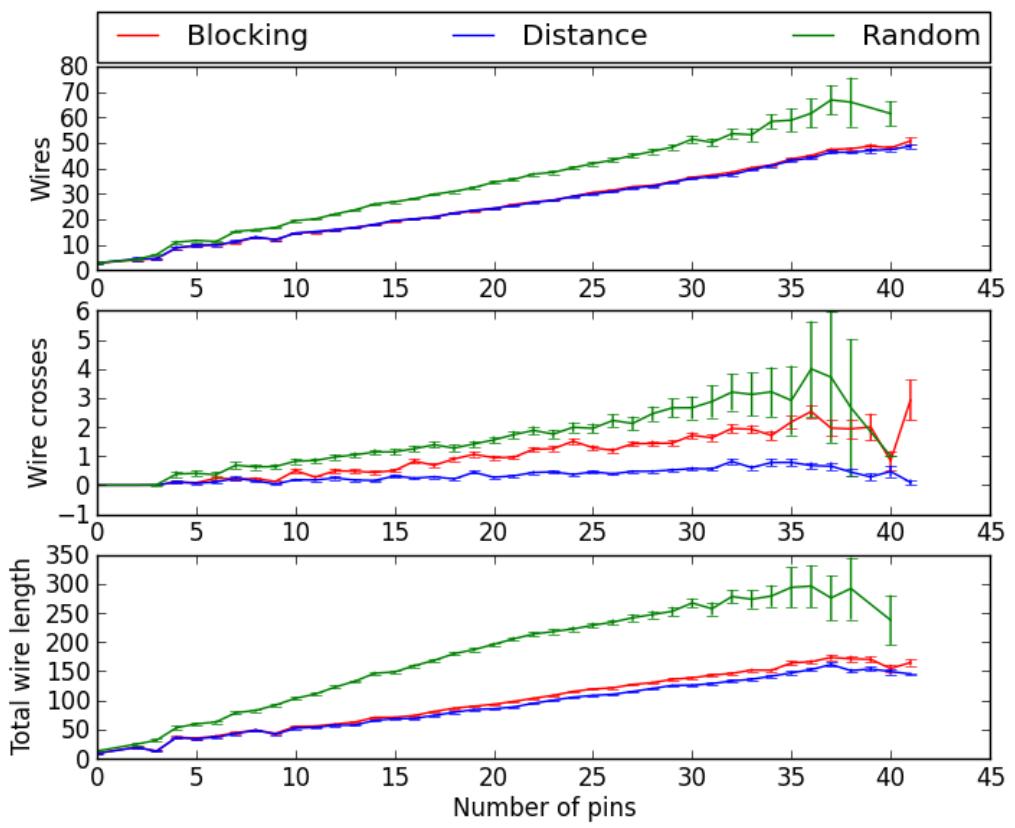


Figure 4-8: Placement method comparison: layout quality trends.

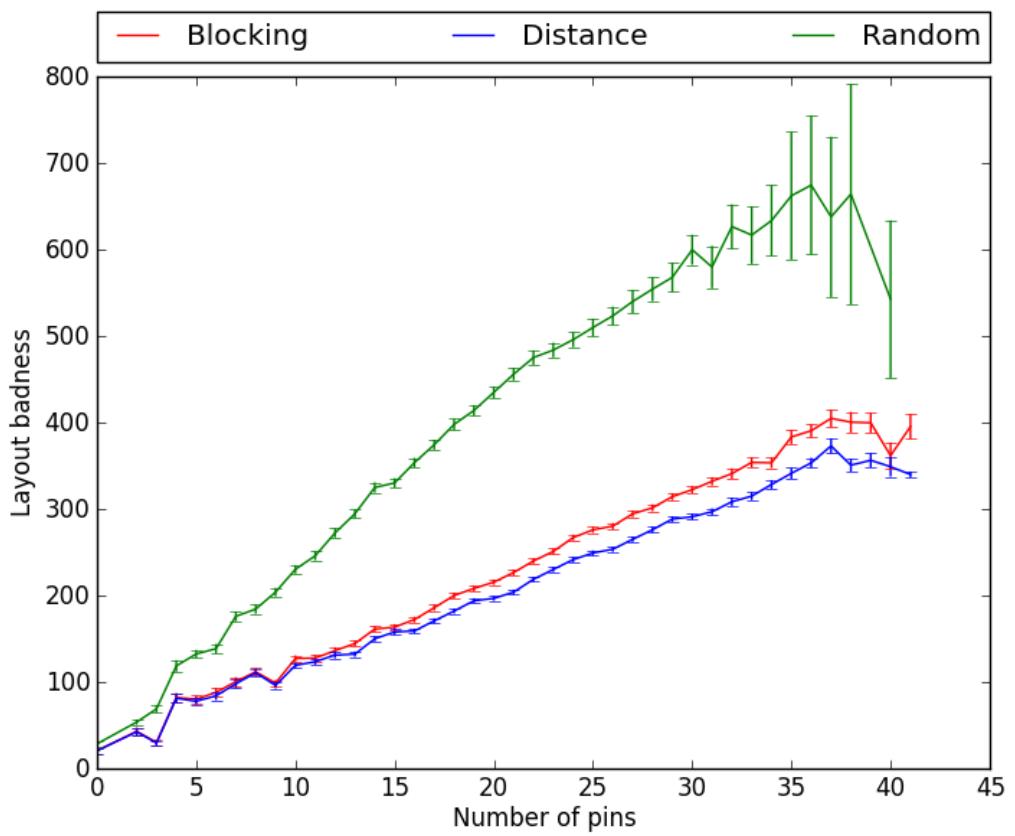
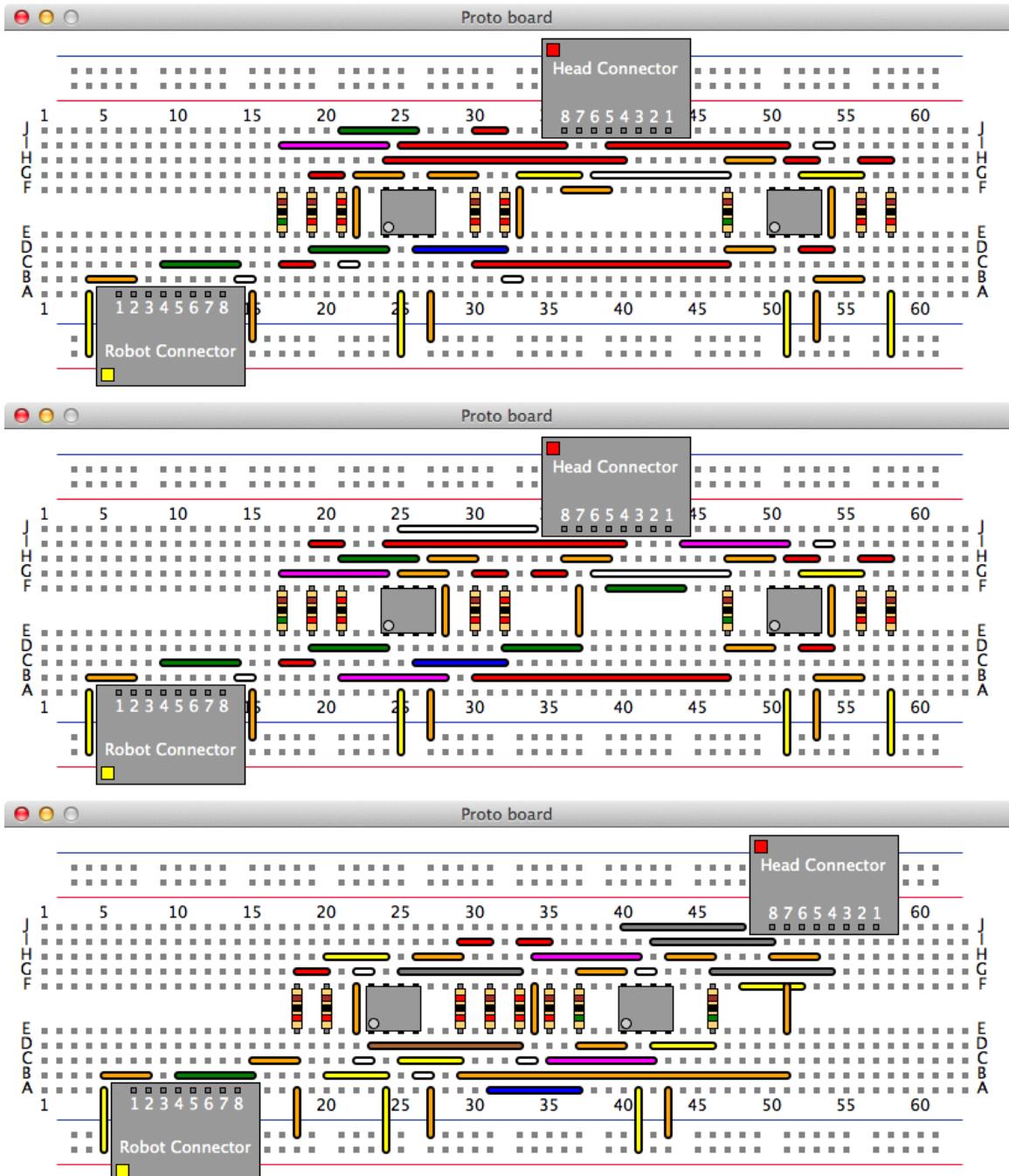


Figure 4-9: Placement method comparison: layout badness trends.



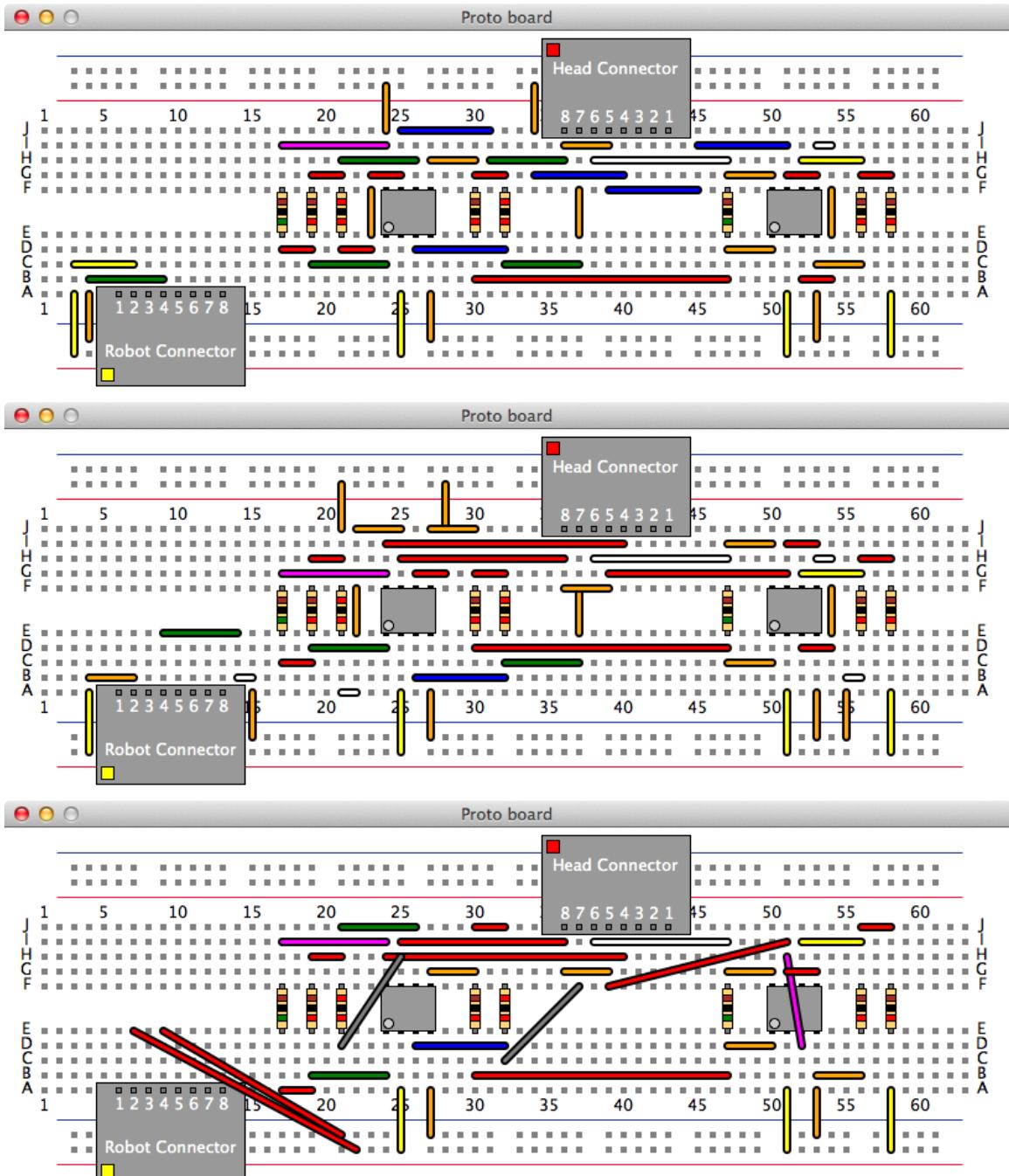


Figure 4-10: All-pairs vs. Per-node, increasing vs. Per-node, decreasing vs. Per-pair, increasing vs. Per-pair, decreasing vs. Straight.

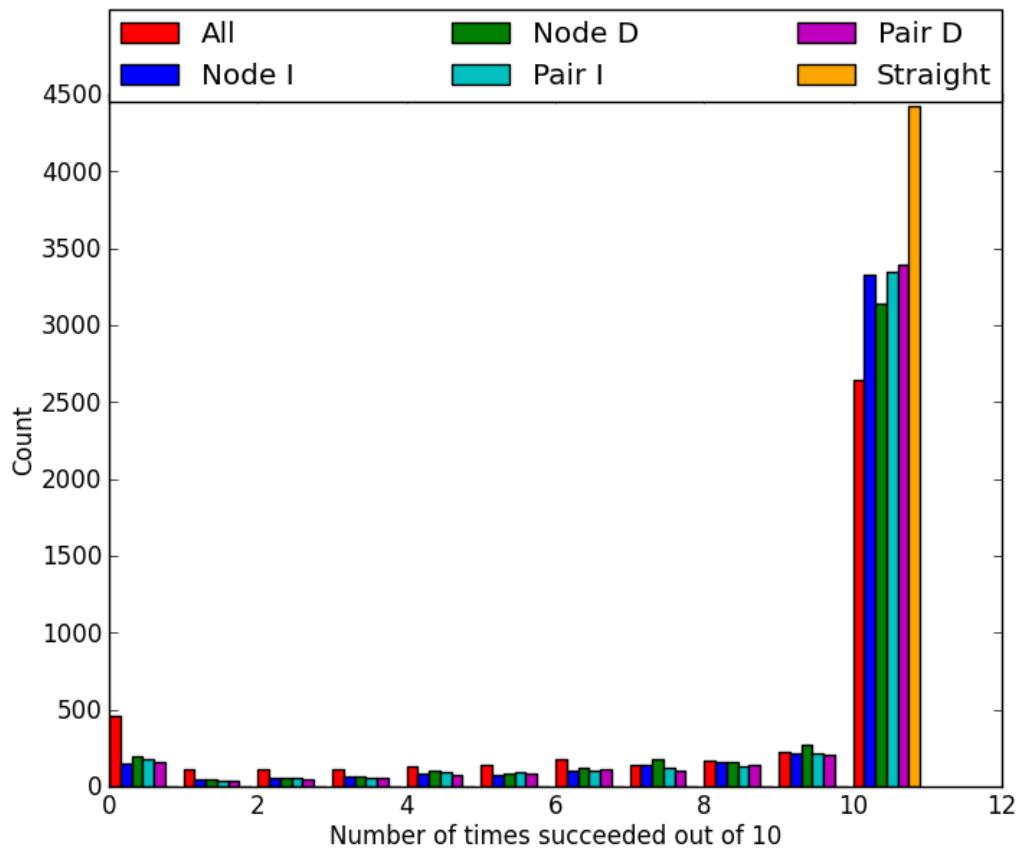


Figure 4-11: Wiring method comparison: success rates.

	Number of times succeeded out of 10										
	0	1	2	3	4	5	6	7	8	9	10
All	458 0.10	114 0.03	111 0.03	112 0.03	127 0.03	145 0.03	177 0.04	139 0.03	172 0.04	227 0.05	2643 0.60
Node I	154 0.03	50 0.01	55 0.01	62 0.01	85 0.02	71 0.02	106 0.02	141 0.03	156 0.04	217 0.05	3328 0.75
Node D	195 0.04	50 0.01	58 0.01	66 0.01	104 0.02	83 0.02	125 0.03	176 0.04	162 0.04	268 0.06	3138 0.71
Pair I	177 0.04	40 0.01	59 0.01	54 0.01	91 0.02	92 0.02	100 0.02	118 0.03	132 0.03	212 0.05	3350 0.76
Pair D	162 0.04	38 0.01	51 0.01	57 0.01	72 0.02	85 0.02	109 0.02	106 0.02	144 0.03	203 0.05	3398 0.77
Straight	0 0.00	0 0.00	0 0.00	0 0.00	0 0.00	0 0.00	0 0.00	0 0.00	0 0.00	0 0.00	4425 1.00

Table 4.2: Wiring method comparison: success rates.

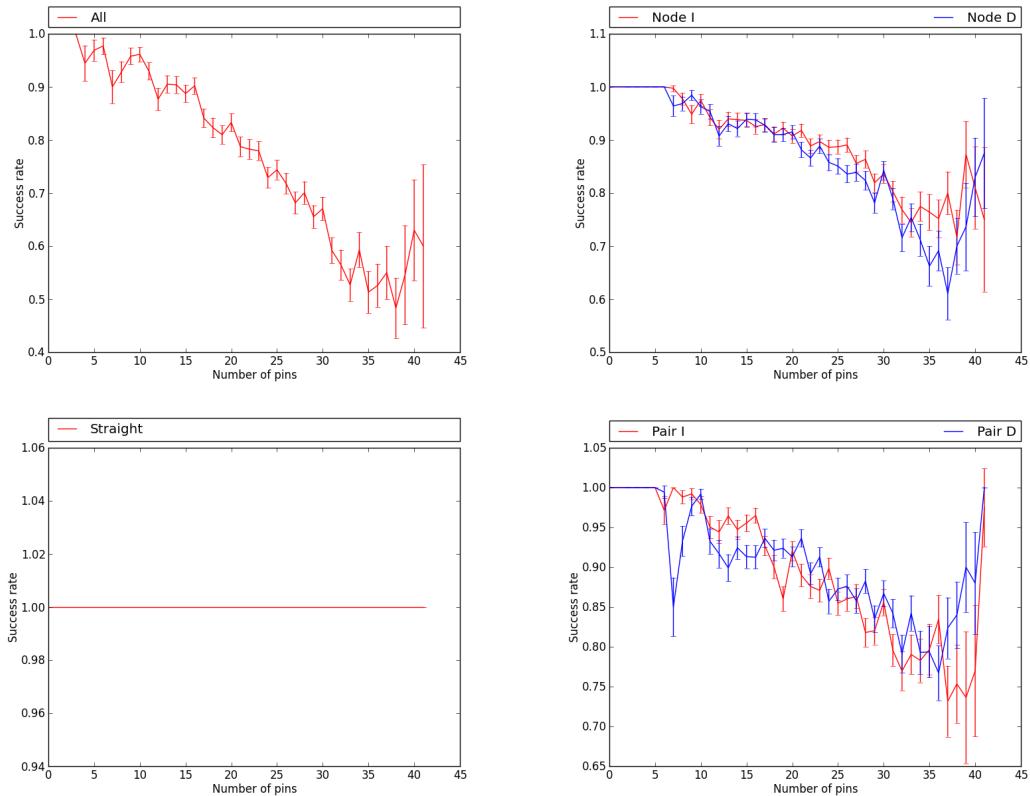


Figure 4-12: Wiring method comparison: success rate trends.

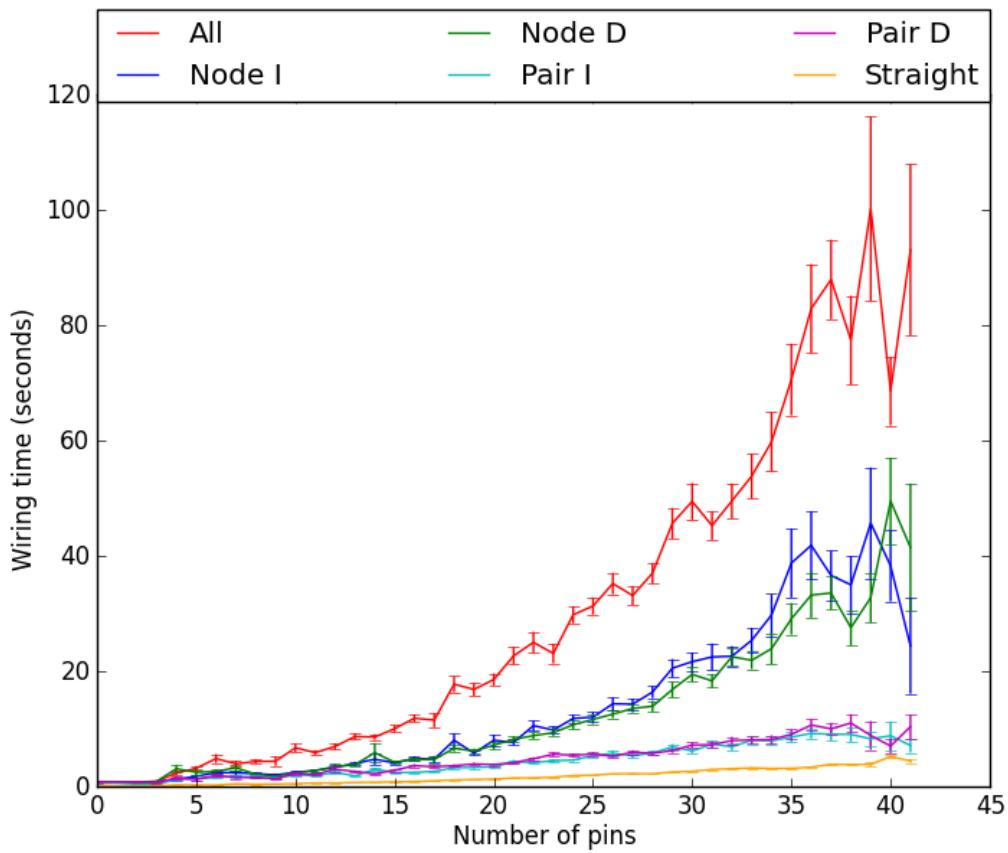


Figure 4-13: Wiring method comparison: wiring time trends.

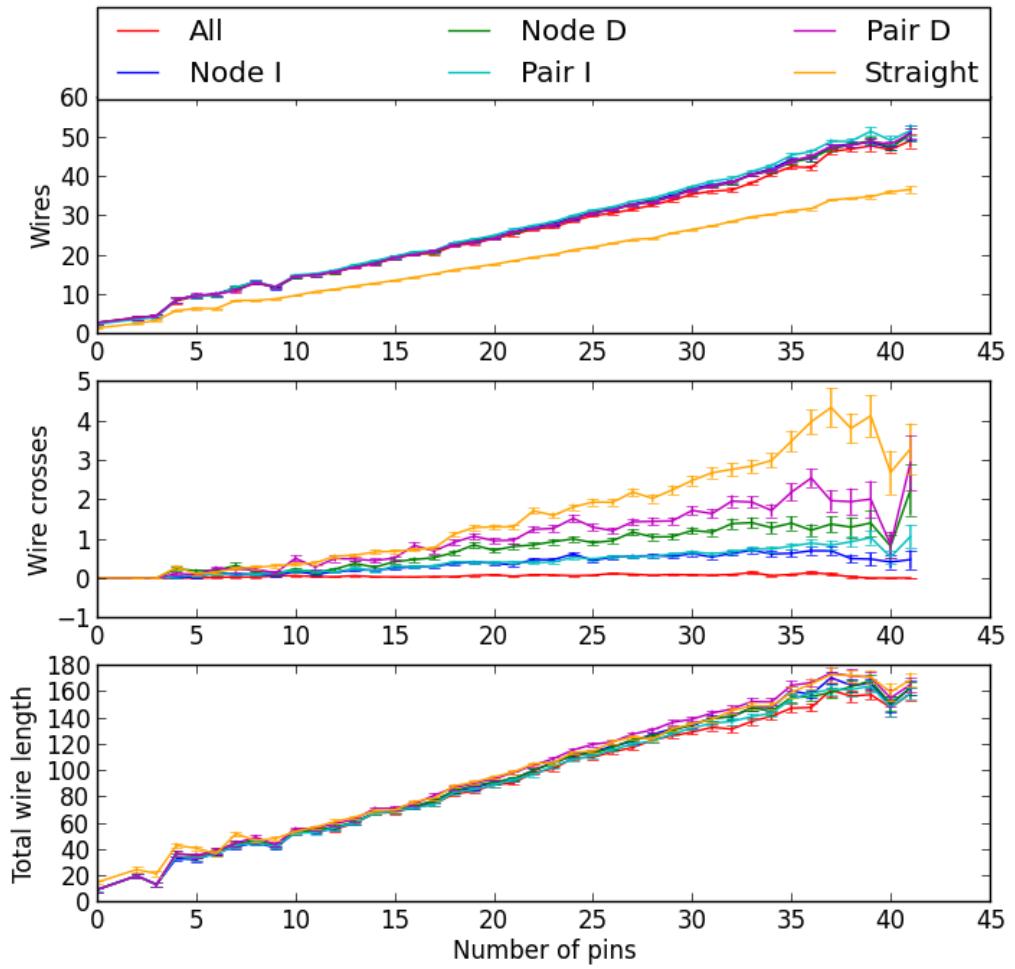


Figure 4-14: Wiring method comparison: layout quality trends.

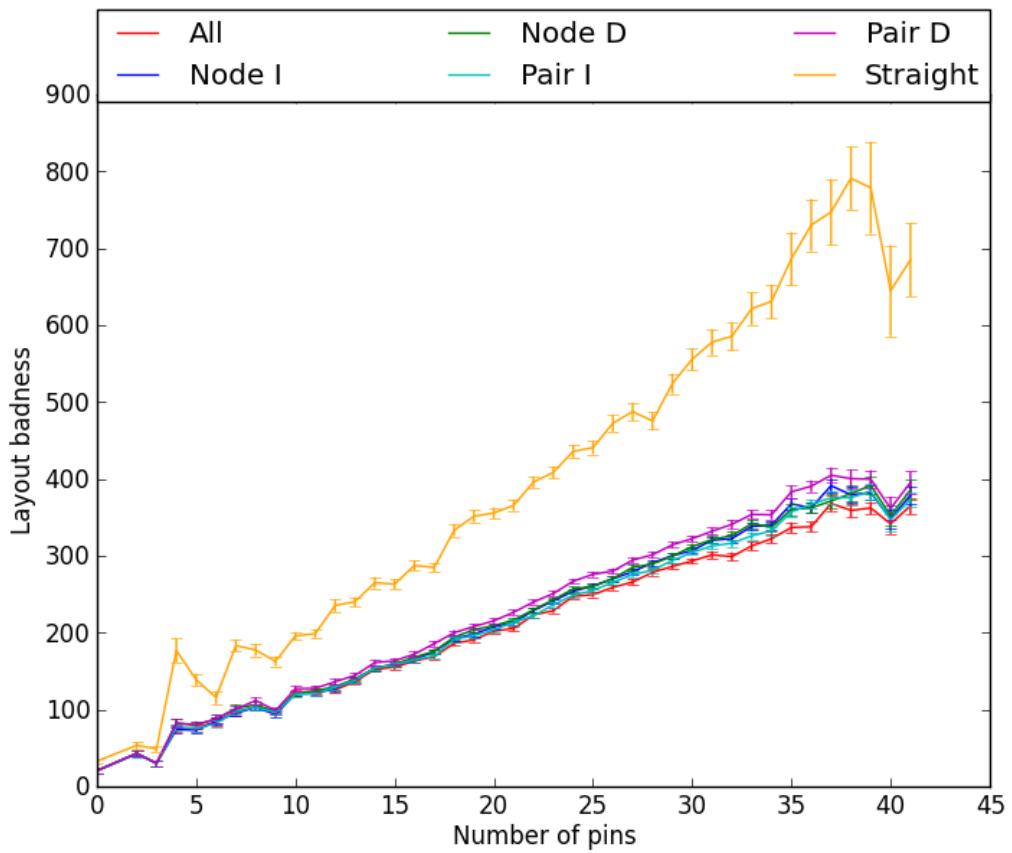


Figure 4-15: Wiring method comparison: layout badness trends.

4.3 Comparing Search Methods

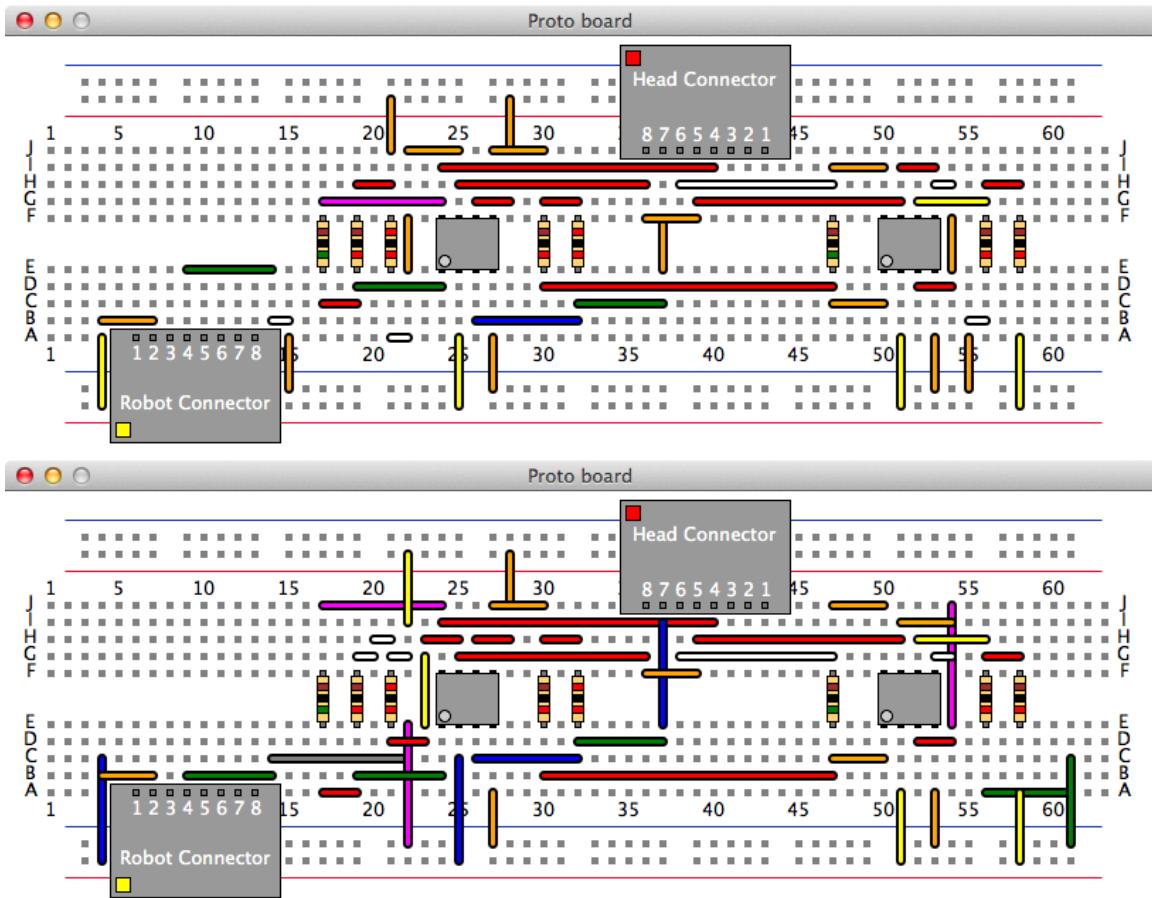


Figure 4-16: A^* vs. Best First Search.

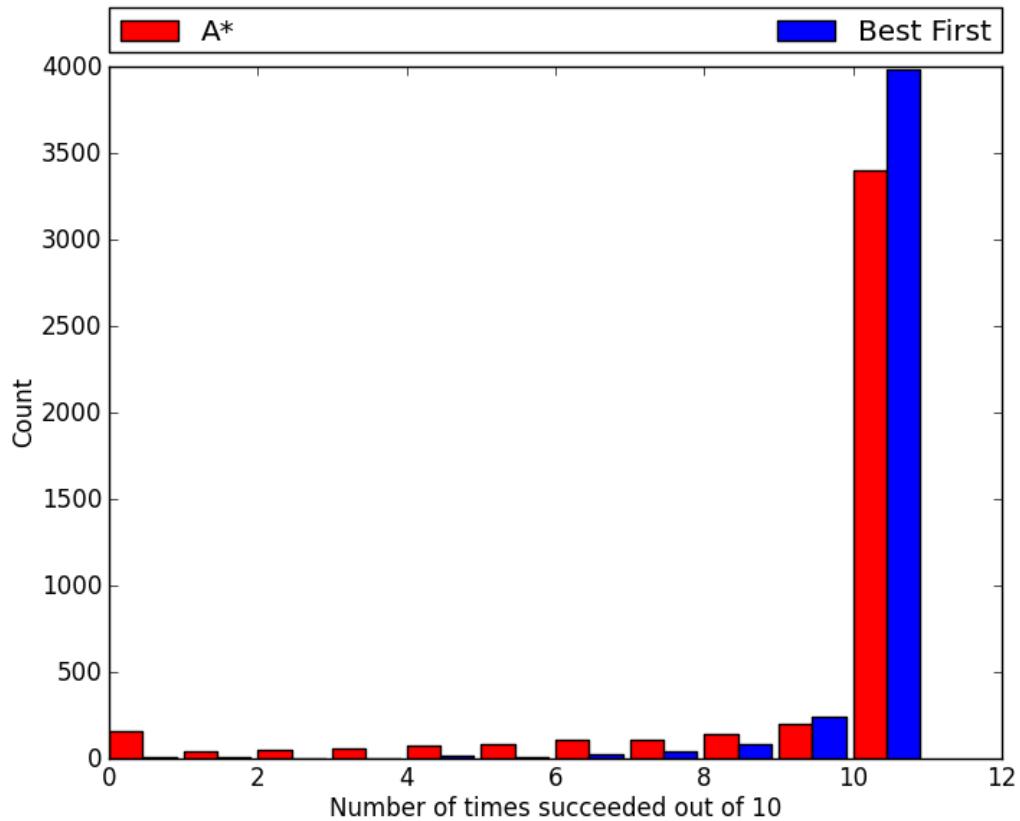


Figure 4-17: Search method comparison: success rates.

		Number of times succeeded out of 10										
		0	1	2	3	4	5	6	7	8	9	10
<i>A*</i>	Count	162	38	51	57	72	85	109	106	144	203	3398
	Probability	0.04	0.01	0.01	0.01	0.02	0.02	0.02	0.02	0.03	0.05	0.77
Best First	Count	6	5	2	1	13	10	29	45	84	245	3985
	Probability	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.01	0.02	0.06	0.90

Table 4.3: Search method comparison: success rates.

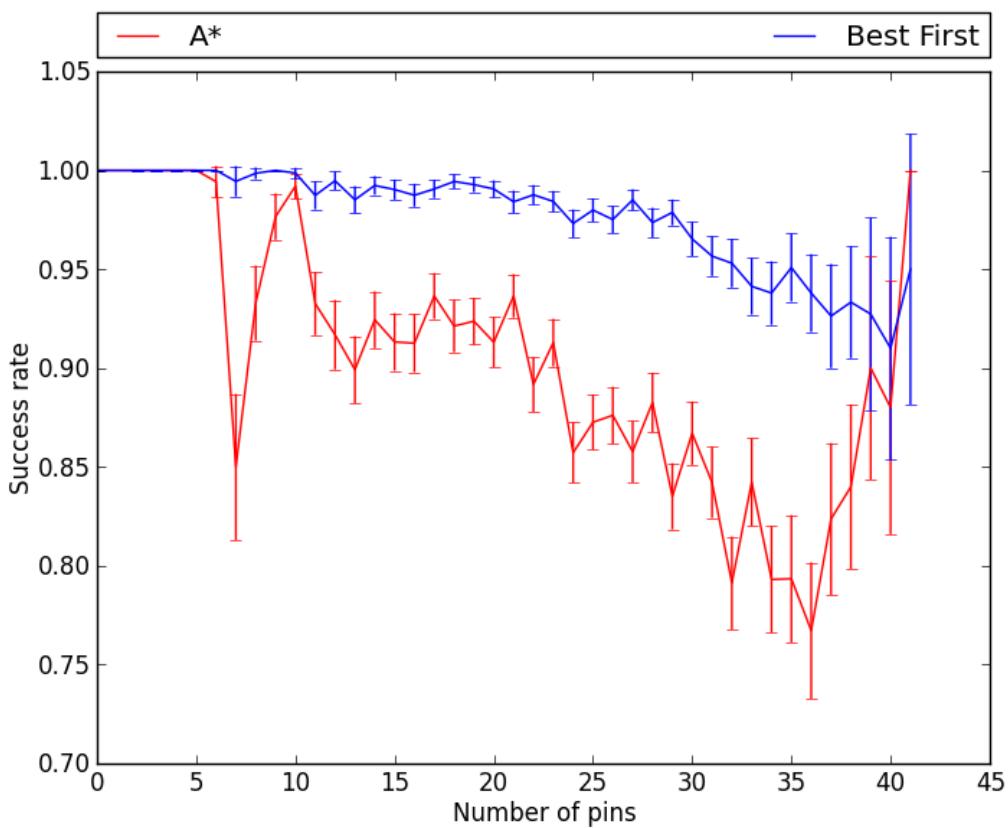


Figure 4-18: Search method comparison: success rate trends.

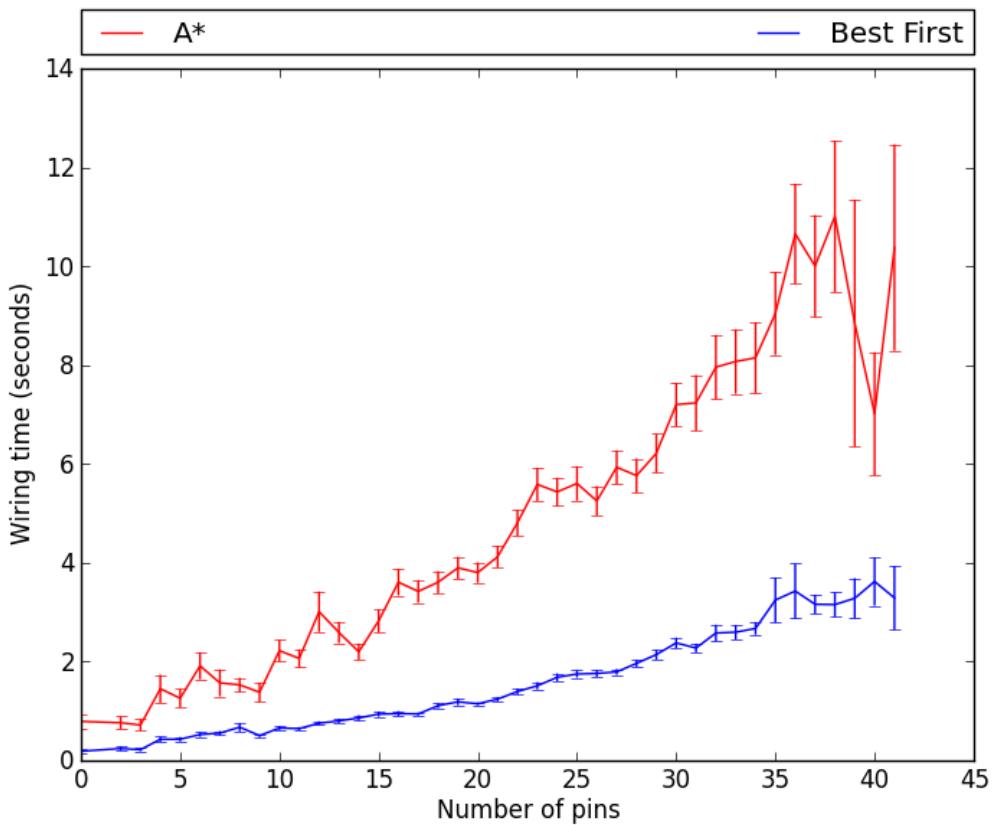


Figure 4-19: Search method comparison: wiring time trends.

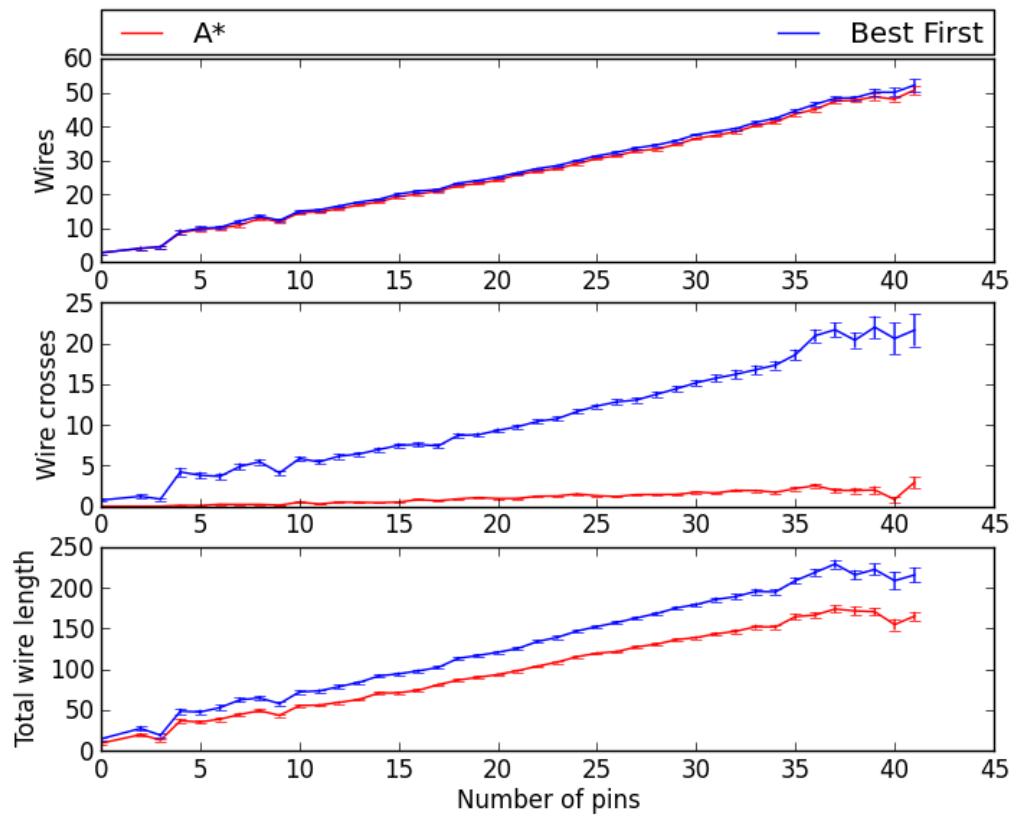


Figure 4-20: Search method comparison: layout quality trends.

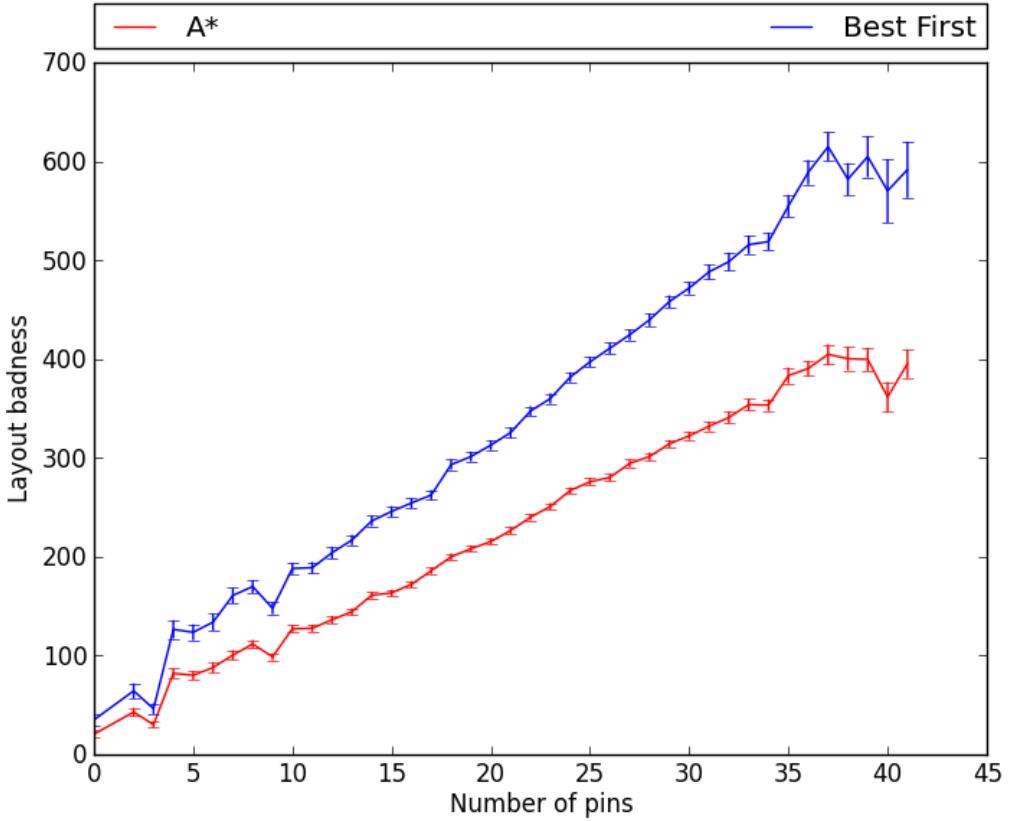


Figure 4-21: Search method comparison: layout badness trends.

4.4 Combined Algorithm

Here we provide data for the combined algorithm presented in Section 3.2.3. To generate this data, we use a different dataset of 4425 schematics. As desired, the algorithm has a 100% success rate. Figure 4-23 gives a breakdown of how the algorithm succeeded. The first four columns correspond to success from one of the four combinations of placement and wiring methods. The last 5 columns correspond to layouts in which none of the four combinations was successful on all pairs of locations and we had to force a certain number of wires. Figure 4-24 gives the average total time taken by the algorithm as a function of circuit complexity. Finally, Figures 4-25 and 4-26 give statistics on the quality of the layouts produced by the combined algorithm

as a function of circuit complexity.

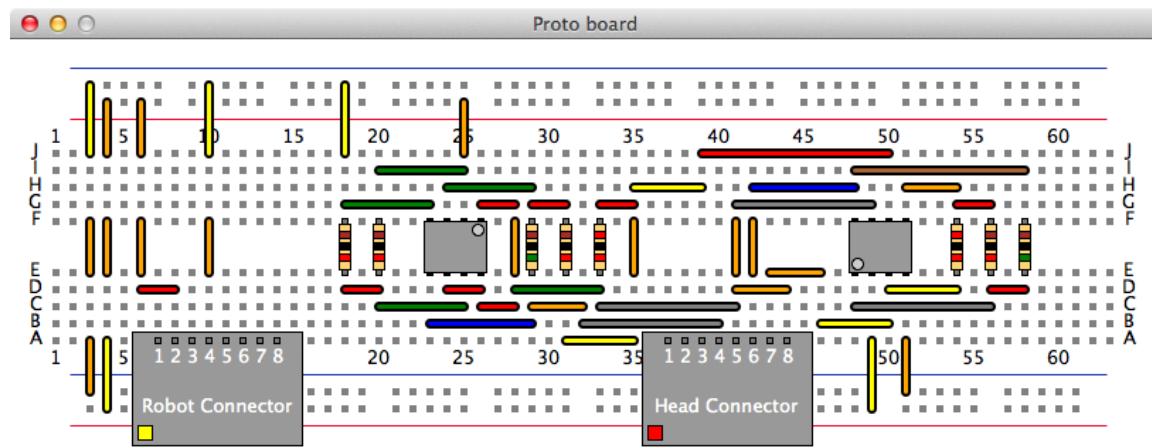


Figure 4-22: Combined algorithm exemplar.

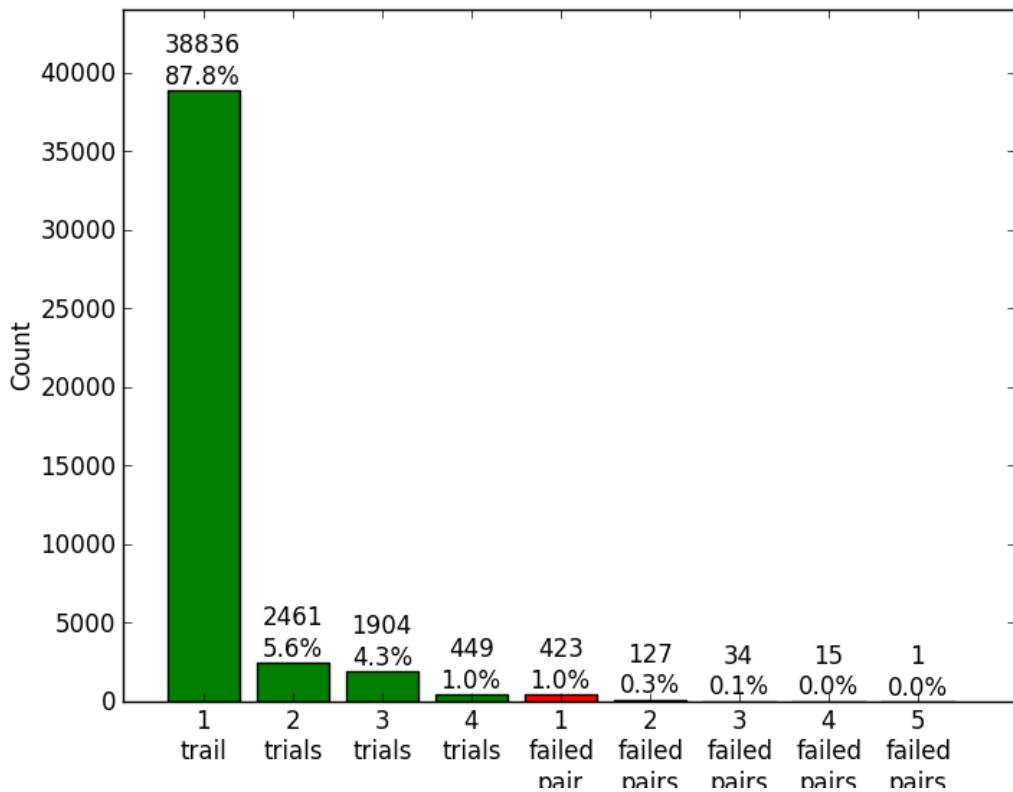


Figure 4-23: Combined algorithm success summary.

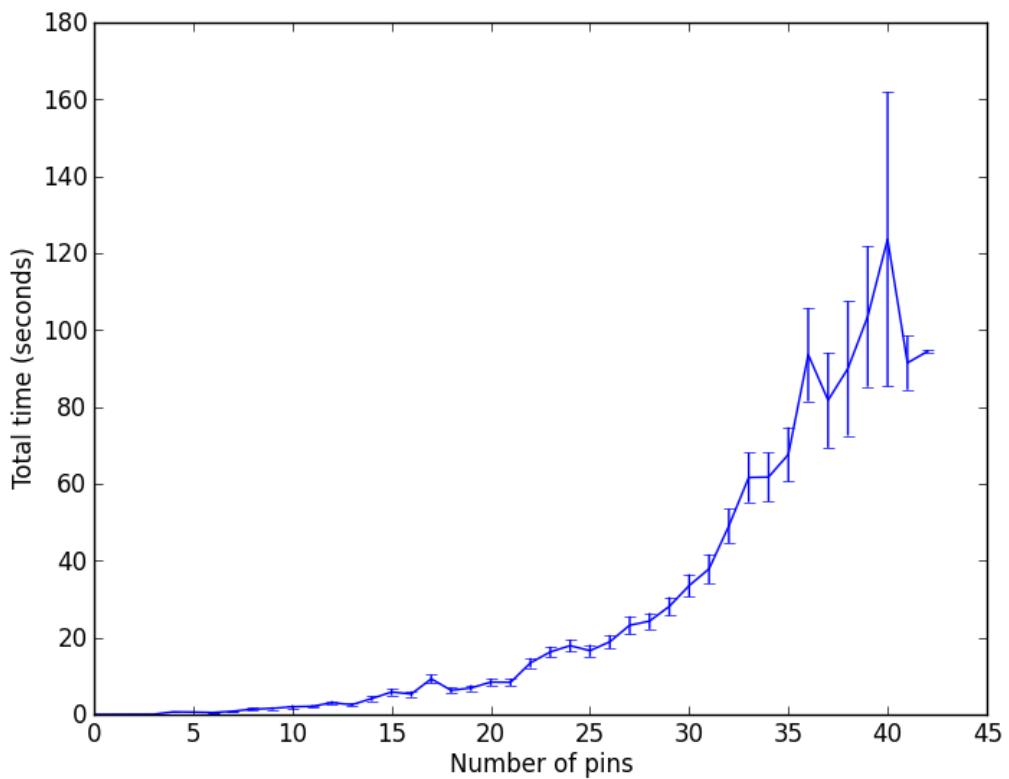


Figure 4-24: Combined algorithm total CPU time trend as a function of circuit complexity.

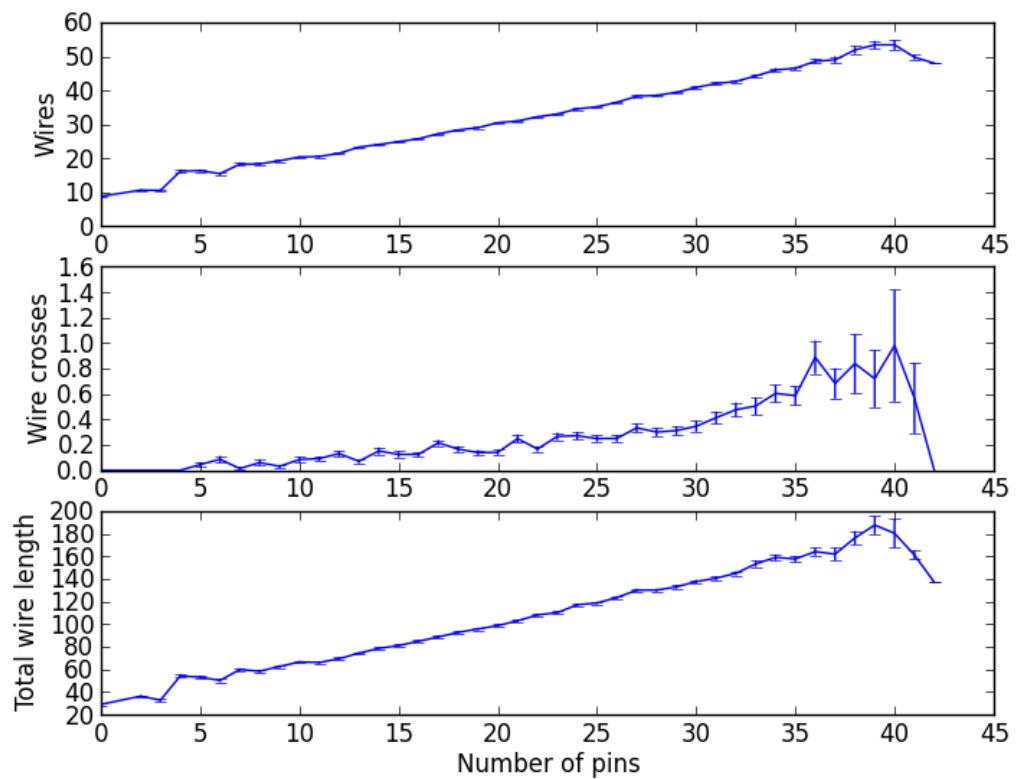


Figure 4-25: Combined algorithm quality trend.

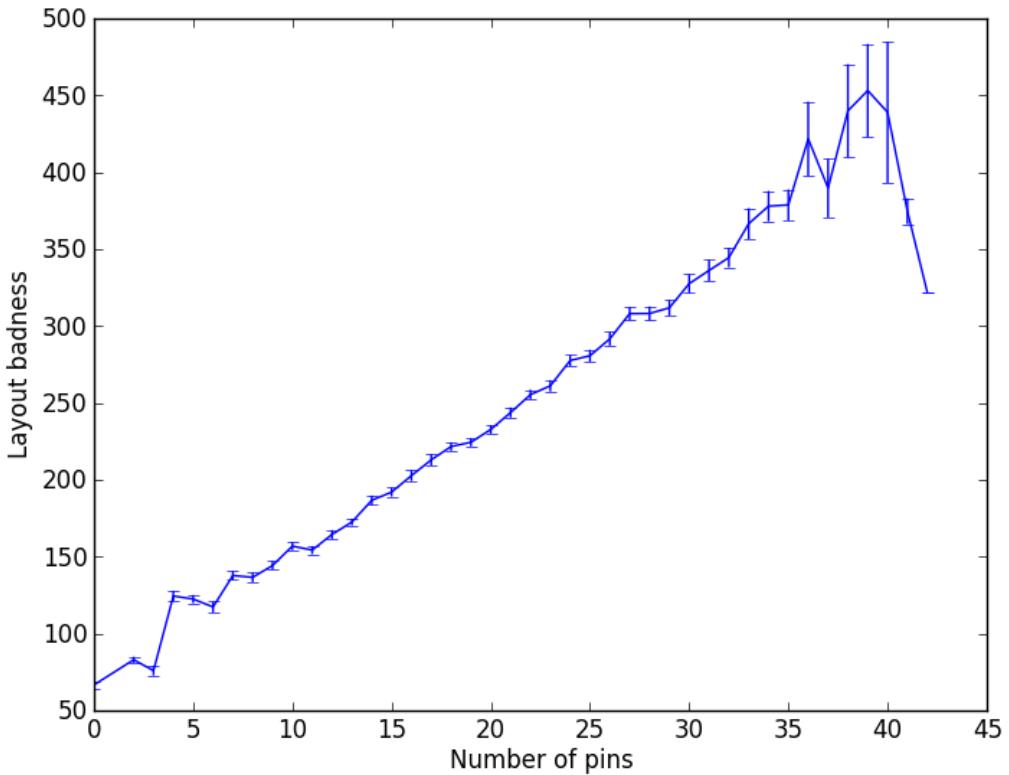


Figure 4-26: Combined algorithm layout badness trend.

4.5 Effect of limiting the number of vertices to expand in A^*

For each of the 5 wiring methods we have looked at, we have run tests on 4425 schematics. We have tested each method 10 times on each schematic. Hence, we have 44250 data points for each of the wiring methods. Here we provide histograms that show the maximum number of expanded vertices in the A^* search among the 44250 runs for each of the 5 methods.

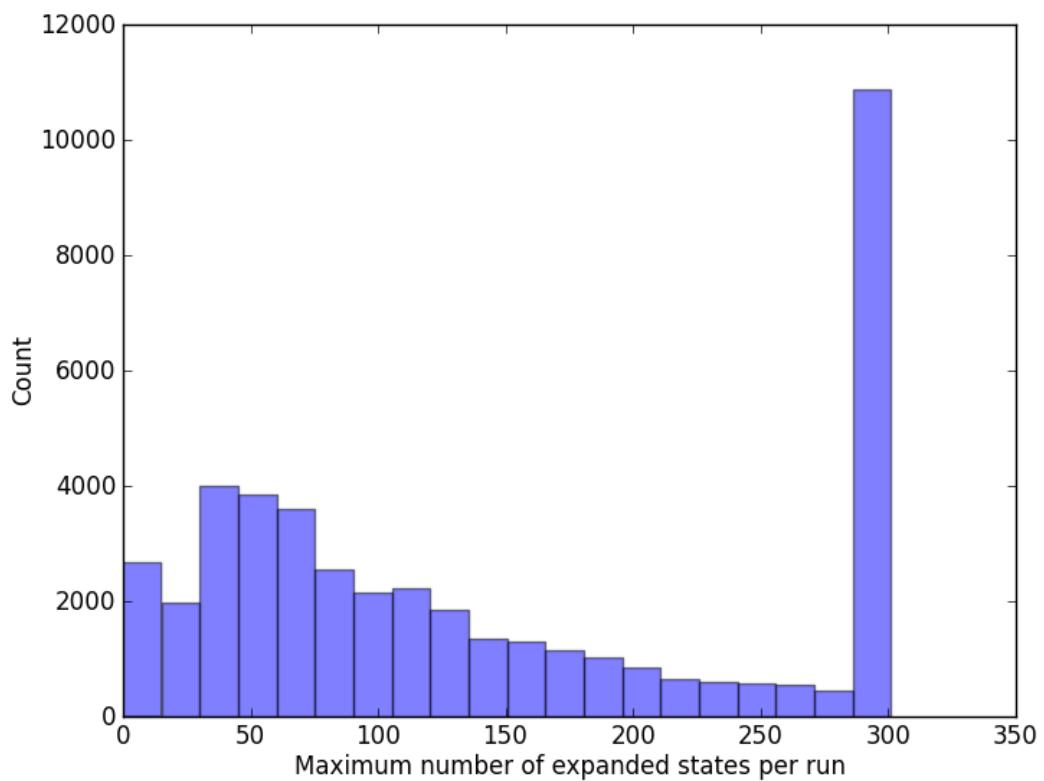


Figure 4-27: Histogram of number of vertices expanded for all pairs wiring.

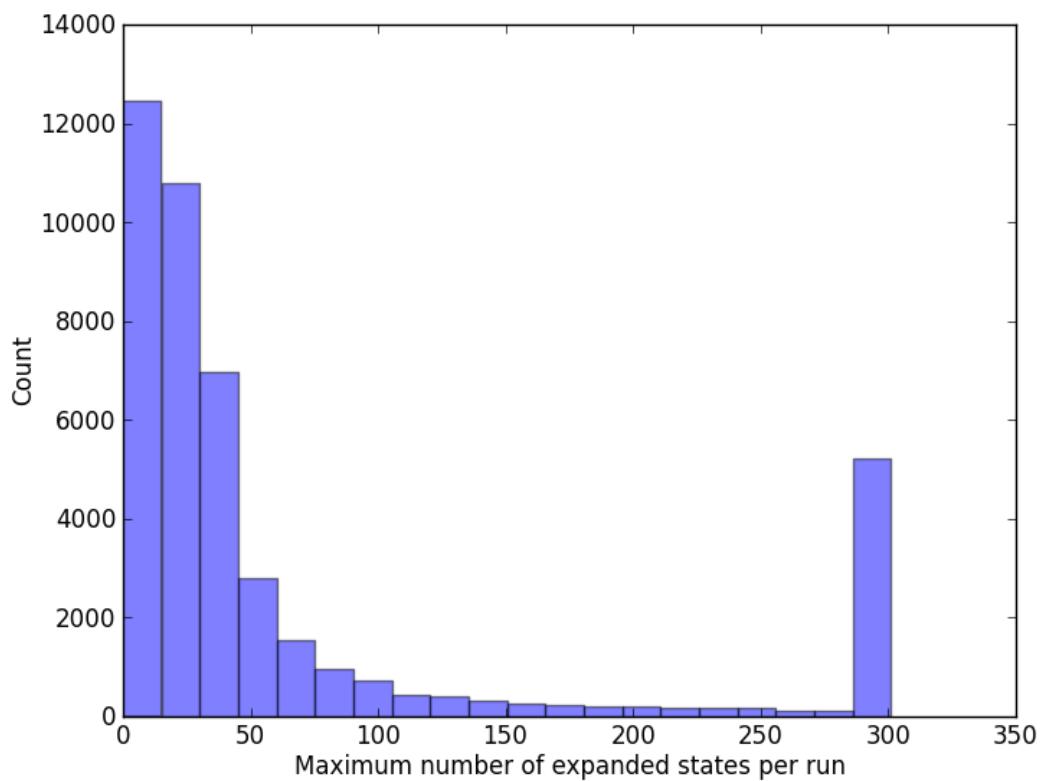


Figure 4-28: Histogram of maximum number of vertices expanded for per-node, increasing wiring.

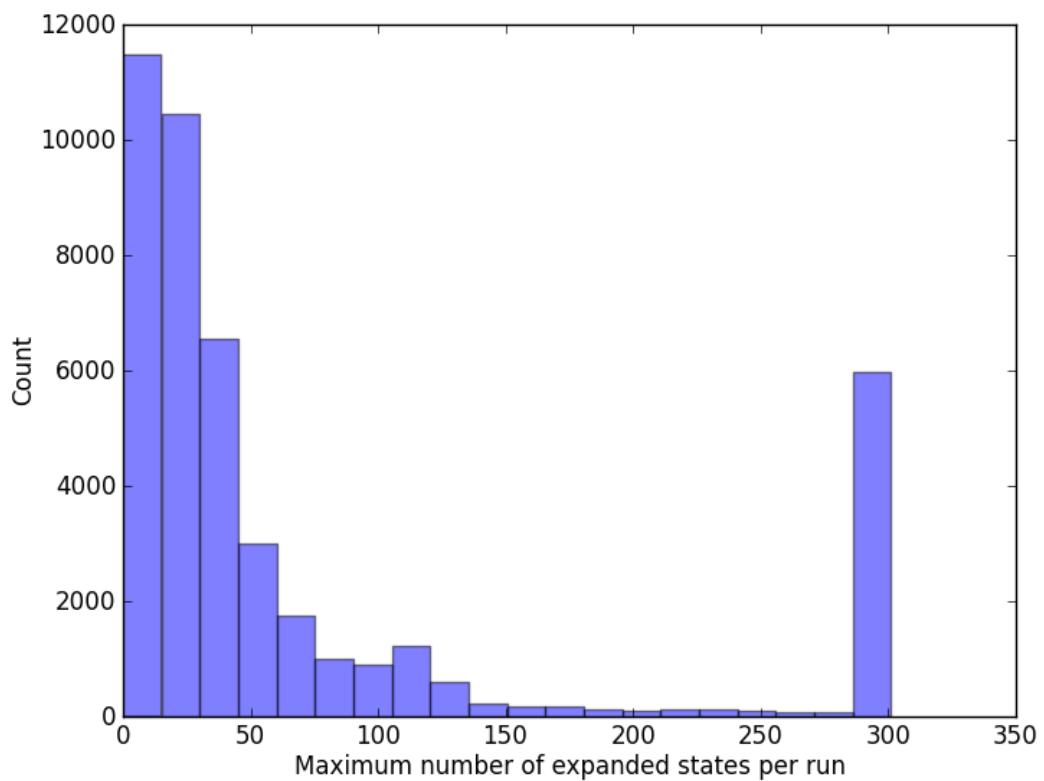


Figure 4-29: Histogram of maximum number of vertices expanded for per-node, decreasing wiring.

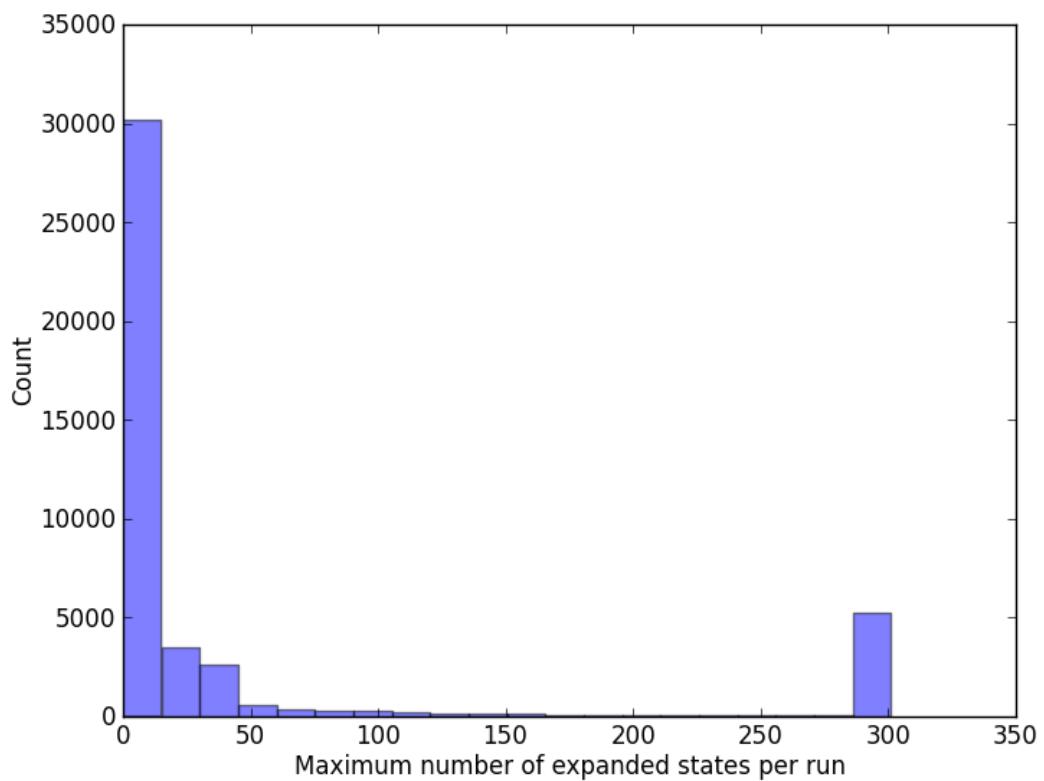


Figure 4-30: Histogram of maximum number of vertices expanded for per-pair, increasing wiring.

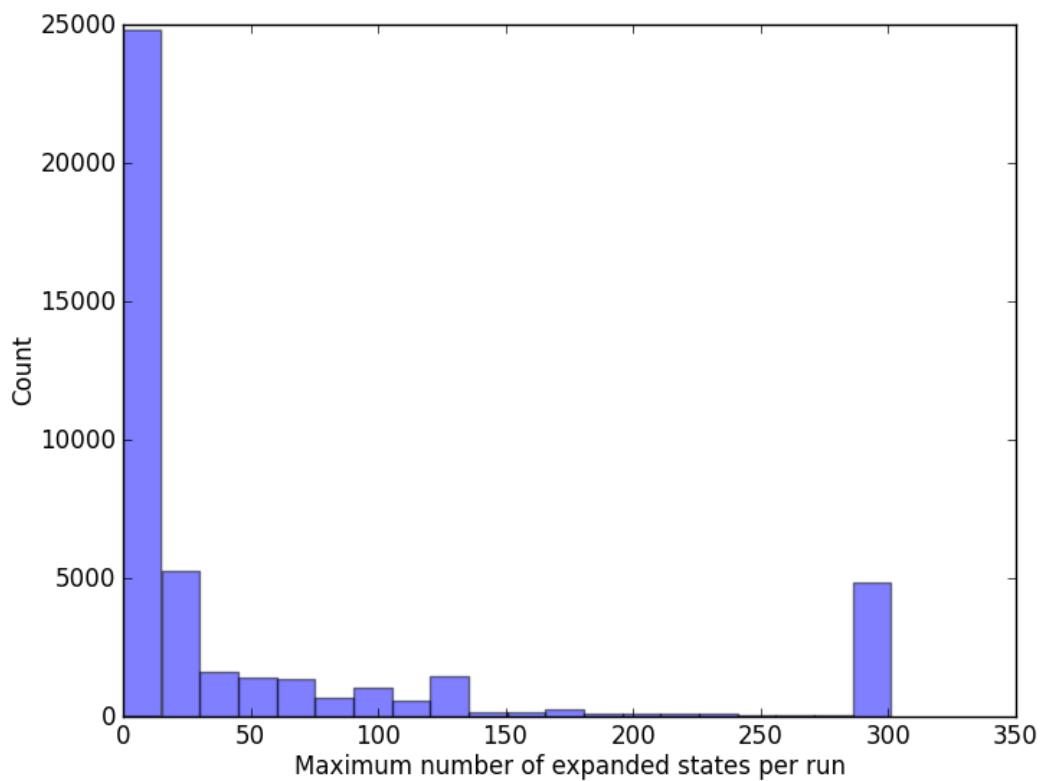


Figure 4-31: Histogram of maximum number of vertices expanded for per-pair, decreasing wiring.

Chapter 5

Discussion

In this section we provide justifications for the choices made in solving the protoboard layout problem, and also detailed analysis of the data presented in Chapter 4.

5.1 Search Space Size

The proposed solution to this problem involves several simplifications and uses of heuristics. This is a result of the fact that the search space we are working with is very large. It is difficult to say exactly how large this search space is, but we can get an idea of its size. Let us just consider the number of ways we can put wires down on an empty protoboard (even in ways that may not make sense from a circuit theory standpoint). Finding this number reduces to finding the number of ways $T(n)$ in which we can choose pairs out of n items. Equation 5.1 gives an expression for $T(n)$.

$$T(n) = \sum_{i=0}^{\lfloor \frac{n}{2} \rfloor} \frac{n!}{i!(n-2i)!2^i} \quad (5.1)$$

In this problem, we have that $n = 830$, the number of available locations on an empty protoboard. Evaluating T at $n = 830$, we get approximately $2.8e1043$. The largeness of this number indicates that doing any sort of exhaustive search will be hopeless.

5.2 Justifying Placement Choices

Resistors

For the sake of simplicity, and to significantly reduce the search space, for every resistor in the schematic, we use one resistor piece placed in the middle strip of the protoboard as shown in Figure 3-2. This choice, i.e. allowing the resistor pieces to only reside in the middle strip of the protoboard, is critical as the resistor pieces can generally be placed at numerous places on the protoboard. With this restriction, there are 63 slots available for one resistor. Without this restriction, there are a total of 763 slots available. The restriction is good when we consider the reduction in the search space size. On the other hand, the restriction is bad when we consider the size of circuits the algorithm can layout. Given that the number of resistors in a typical 6.01 circuit is very small, this restriction proves to be very useful.

Op-amps

Op-amps are the trickiest components to handle because each op-amp package put on the protoboard contains two op-amps within it. Equation 5.2 presents an expression for the value $f(n)$, the number of different ways to package together n op-amps. For example, if we have 2 Op Amps, we can either use one op-amp package for each, or put them both in the same package, which we can do in one of two different ways. Hence, $f(2) = 3$. To get a sense of how many different packagings are possible, Table 5.2 gives the values of $f(n)$ for various n .

$$f(n) = \sum_{k=0}^{\lfloor \frac{n}{2} \rfloor} \frac{n!}{k!(n-2k)!} \quad (5.2)$$

Our placement approach explores all possible ways of packaging up the op-amps. We do this because the typical 6.01 circuit contains no more than 6 op-amps, and so we are tasked with exploring at most 331 alternatives, which doesn't happen to be too computationally intensive. If the algorithm was meant to handle a larger number of op-amps (for example 10), then this approach would be silly, and we would have

n	$f(n)$
1	1
2	3
3	7
4	25
5	81
6	331
7	1303
8	5937
9	26785
10	133651

Table 5.1: Number of ways of packaging together n op-amps for various values of n .

to resort to perhaps considering the packagings that require the fewest number of op-amps to make a choice.

5.3 Explaining the Results

Chapter 4 presented quantitative data to compare the various alternatives we have in solving the protoboard layout problem. Here, we will analyze that data and give reasonings for why we obtained the results that we obtained.

5.3.1 Comparing Placement Methods

In Figure 4-5 and Table 4.1 we see that the distance method exceeds the blocking method in number of circuits solved 10 times out of 10 as well as in the number of circuits solved 0 to 2 times out of 10. On the other hand, the blocking method exceeds the distance method in number of circuits solved from 3 to 9 times out of 10. Overall, the blocking placement method is slightly more successful than the distance placement method. Despite this difference, we note that the two alternatives have very close success rates. We do note, however, that both methods are significantly more successful than using a random placement. As a function of circuit complexity, Figure 4-6 suggests that the two alternatives have almost identical success rates. As we would expect, we certainly observe that as the complexity of the circuits increases,

success rate generally decreases for both of the placement methods. Once again, we observe that the distance and blocking based placement methods are much more successful than random placement, especially as the circuit complexity increases. The curves in Figure 4-6 seem to shoot back up at the far end of the figure, but this is a result of the fact that there are very few circuits at that end of the figure, on which the algorithm happened to be consistently successful.

When we consider wiring time as the basis for comparison, we observe from Figure 4-7 that, once again, the two methods are very similar, with random placement being markedly worse than both. We see that the distance method generally takes slightly less wiring time than the blocking method, but the difference between the two is almost negligible. As we would expect, we certainly see that as the complexity of the circuits increases, the amount of time spent by the wiring step also increases. As we did for the success rate trends as a function of circuit complexity, we observe that there are outliers at the far end of the figure due to a very small sample of the most complex circuits in the randomly generated schematic dataset.

Finally, let us look at layout goodness as the basis for comparison. Figure 4-8 presents graphs that compare numbers of wires, numbers of wire crosses, and total wire lengths. Figure 4-9 shows the trend of layout badness computed using our metric as a function of circuit complexity. We first observe that the number of wires used by the two methods are almost identical. As the complexity of the circuit increases, we see that the blocking method uses more wires than does the distance method, but by and large the values are very comparable. When we look at the number of wire crosses in the layouts, we see that the blocking method consistently results in more wire crosses. Similarly, when we look at the total length of wires used in the layouts, the blocking method exceeds the distance method consistently, with the difference getting higher as complexity increases. This can be explained by the fact that the blocking method may require elaborate connections, especially as the circuits get more complex while the distance method is tuned to require wirings between pairs of locations that are closer together. Finally, we do see that the random placement method, per our badness metric, produces much worse layouts than the other two

methods even when it does succeed.

It is difficult to conclusively pick the best placement method from these results. What we can certainly tell is that the distance based method and the blocking based method are both significantly better than random placement.

5.3.2 Comparing Wiring Methods

To compare the wiring methods, let us start with the success rate for each method. Figures 4-11 and 4.2, and Table 4.2 provide the appropriate data. The very first fact we observe is that the all-pairs method has a much smaller success rate than all of the other alternatives, especially as circuit complexity increases. The reason for this, in large part, is the fact that we impose a cutoff on the number of nodes to expand in the search (TODO: this must be discussed in Methods). As we are using just the one search to connect all pairs of locations, the search cutoff would be expected to have more of an effect on the all-pairs method. The other four alternatives have very comparable success rates.

Next we look at Figure 4-13 to compare wiring times for the five methods. Once again, we observe that the all-pairs methods takes significantly more time than the other methods. When attempting to connect all pairs in one search, the method searches for an appealing layout, which may require searching through a large number of alternatives, especially for more complex circuits. Hence, we would expect the all-pairs methods to generally take more time than the other alternatives. We also observe from Figure 4-13 that the wiring times for the two per-node methods are comparable, and that the wiring times for the two per-pair methods are also comparable, but that the per-node wiring times are generally bigger than the per-pair wiring times. This trend is also expected as the per-node methods attempt to connect multiple pairs of locations at once while keeping the layout pleasing, and this generally requires searching through more alternatives than connecting each of the pairs of locations individually. It is important to note that per-node, increasing generally takes more time than per-pair, decreasing, and also that per-pair, increasing generally takes more time than per-pair, decreasing.

Finally let us look at Figure 4-14 to compare the quality of the layouts produced by the five alternative wiring methods. First, as we observed in the placement method comparison, we see that there is very little difference in terms of number of wires used and the total wire length. However, there are significant differences in the number of wire crosses. We see that the all-pairs method generates layouts with much fewer wire crosses than the other methods. This is completely expected since we run one search to connect all pairs of locations while attempting to keep the layout as nice as possible. Conversely, the per-pair, decreasing and per-node, decreasing methods result in the most number of wire crosses. It is very interesting to note that the per-node, decreasing method produces more wire crosses on average than the per-pair, increasing method. Here we observe that the order in which we consider pairs of locations has a telling effect on how good the layouts will be. In essence, connecting the harder pairs of locations generally produces more wire crosses.

What we have observed is that while the all-pairs method is the least successful method and the one that generally takes the longest among the five, it generally produces the best layouts when it does succeed. On the other hand, the alternatives that break down the problem into smaller pieces succeed more often and finish more quickly while producing worse results. Furthermore, the more finely we breakdown the problem, the faster the overall algorithm. Lastly, ordering subproblems from hardest to easiest has the effect of making the overall wiring step complete faster but producing worse results than the reverse order and not necessarily getting a markedly better success rate.

5.3.3 Comparing Search Methods

We now compare our two alternative search algorithms: *A** and Best First Search. Let us start by comparing success rates. Figures 4-17 and 4-18 and Table 4-17 present the appropriate data. We clearly observe that Best First Search is much more successful than *A**. 98% of the test circuits were solved at least 8 times out of 10 when we used Best First Search, versus 85% when we used *A**. This result is not surprising because Best First Search hungrily hunts for layouts that satisfy the connection requirements

without caring for the aesthetics of the layouts. Hence, Best First Search is much less susceptible to the 300 states to expand restriction than A^* . The fact that the quality of the results we get from Best First Search are worse is clearly evident from Figure 4-20. Most importantly, the number of wire crosses in the layouts produced by Best First Search are markedly greater than the number of wire crosses in the layouts produced by A^* . We also observe that the total wire length is greater when using Best First Search. The fact that Best First Search settles for any layout that satisfies the connection requirements suggests that it should finish more quickly in addition to being more successful. Figure 4-19 supports exactly this expectation.

Our choice of a search algorithm forces us to consider a trade-off between quick success and quality. If we choose Best First Search, most runs will be successful and terminate quickly, but will produce very poor results. If we choose A^* , not as many runs will be successful, and the successful runs will take longer to terminate, but we will get much better layouts.

5.3.4 Putting Them All Together

Now we will discuss why the combination algorithm given in Algorithm 2 is structured the way it is, and we will also discuss the data we obtained for the combined algorithm. Recall that the combined algorithm makes four attempts at generating a complete layout. We try both placement methods, the distance method first and the blocking method second. We try the distance method first because it tends to generate layouts with fewer crossing wires and smaller total wire length. We use per-pair wiring, and consider both orders of doing the wiring, increasing order first and decreasing order second. We use per-pair wiring because it takes considerably less time than either per-node or all-pair wiring, and neither of the other wiring methods has a better success rate. We try increasing order first because that tends to generate layouts with fewer crossing wires than decreasing order.

Let us now consider the data we obtained for the combined algorithm. Firstly, we saw that the combined algorithm had a 100% success rate, which is critical when we consider the fact that students will be using a tool that almost never fails. Next,

Figure 4-23 presents very encouraging results as well. Not only is the algorithm 100% successful, on the test dataset of 4425 schematics, with 10 runs carried out on each schematic, the algorithm succeeded 98.6% of the time without having to put down forced wires. The algorithm was required to put down more than 2 forced wires only 0.1% of the time. From Figure 4-24 we see that as the circuit gets more complex, the amount of time the algorithm takes sharply increases. This is due to a more difficult placement task (especially as we have more op-amps) as well as a more difficult wiring task. An important aspect of this Figure is that the maximum point on the plot occurs at less than 180 seconds. This indicates that on the test dataset, the algorithm took at most about 3 minutes to run. This is encouraging from a practical stand point because 3 minutes is not terribly long for a student to have to wait for a layout to be generated. Finally Figure 4-25 presents trends of quality as a function of circuit complexity. Most importantly, we observe that the average number of wire crosses never exceeds 1.

5.4 Further Work

5.4.1 Treating Resistors as Wires

Our current solution treats resistors as one of the circuit components that we work with. However, resistors have the special property among the components that they look exactly like wires of length 3. In fact, the length need not even be specified. We can think of resistors as wires of a fixed range of lengths. So one idea we may attempt is to treat resistors in the wiring step of the algorithm instead of the placement step. The search that we do in the wiring step would now need to be more elaborate. Not only do we need to keep track of pairs of locations on the protoboard that need to be connected, but we also need to know whether to put a resistor between the two locations. In the latter case, we essentially have the restriction that one of the wires we use to connect the pair of locations needs to be of a length that can fit a resistor.

While this idea appears very promising, there are a few difficulties we may face

in implementing it. First, if there is any node in our circuit that is only connected to resistors (and no other circuit components), then that node will be unrepresented on the protoboard at the end of the placement step. One rather unsatisfying solution to this problem may be reserving an empty column on the protoboard for the node that we can use in the wiring step. A better solution may be discovering the best places for the node as we are placing down resistors in the wiring step, but this solution would certainly make the search we carry out more complicated. Second, we will need a new kind of heuristic for our *A** search that takes resistors into account. The first idea that comes to mind may be highly penalizing an absent resistor. Even with an amended heuristic, there are cases we can imagine where the search may have a difficult time getting an answer, for instance a pair of locations that are right next to each other that must be connected by a resistor.

5.4.2 Building Layouts Similar to Previously Generated Layouts

One of the weaknesses of our tool is that it may generate different layouts for the same circuit schematic. This is not a result of an explicit randomization in our algorithm, but rather a side effect of some data structures we use such as Python sets and dictionaries. A similar problem is that a slight change in the circuit schematic may result in a completely different layout. It is, therefore, very important that students in 6.01 be confident that they designed the right circuit before taking the time to build their circuit. This is where the simulation capabilities of our tool come in to play. As an alternative solution to this problem, it may be very useful to have our tool remember the last placement it used and try to produce a new placement as similar to the last one as possible in the case that the circuit was not dramatically changed between requests for a layout.

5.5 Remarks

The results we have gotten for the final algorithm are indeed very encouraging. It has a 100% success rate and produces layouts that, by our metric, are very good. The data we have collected suggests that we can put the tool in front of students, and let them never worry about layout and only worry about circuit design. If this tool saves 1 lab hour per student per semester (time the student would have spent producing layouts), then we are looking at a savings of about 200 6.01 lab hours per semester, which is fantastic success.

Appendix A

Schematic Drawing GUI

One of the aims of this project is making it easy for students to describe the circuit they have in mind to a computer. To that end, it is important that the schematic entry tool be intuitive and easy to use. In this section, I will describe the capabilities and features offered by the schematic entry Graphical User Interface. Figure A-1 presents the GUI containing a sample schematic. The figure depicts the four important sections of the GUI: the board, the palette, the analysis section, and the cursor toggle section.

A.1 Palette

The palette (item 3 in Figure A-1) offers all of the circuit components that can constitute a 6.01 circuit. Clicking a circuit component on the palette spawns a new component of the same type on the board right above the palette. This component can then be used in the circuit construction. The "Robot" and "Head" buttons on the palette spawn multiple parts at once corresponding to the multiple parts contained within each item. The robot connector contains pins for power and ground as well as for analog input pins and one analog output pin. The head is composed of a motor, a motor pot, and two photosensors. Figure A-2 demonstrates these last two circuit components.

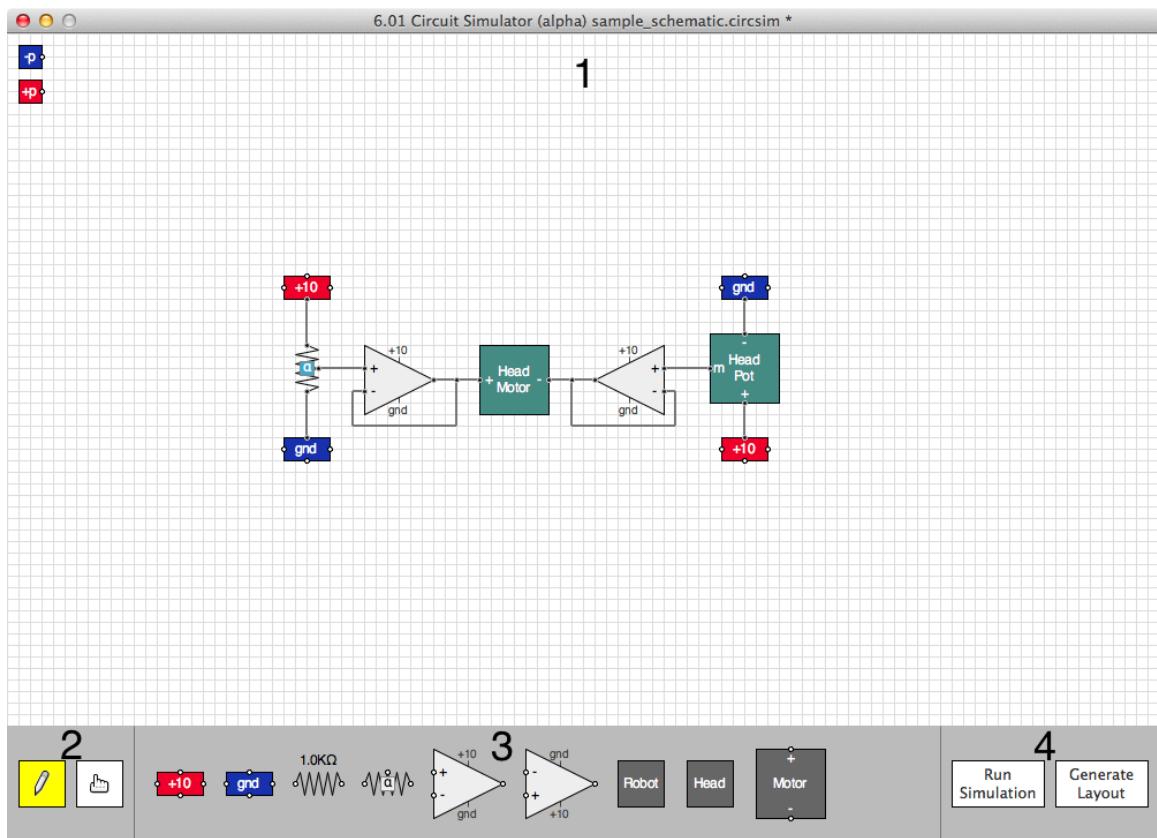


Figure A-1: Parts of the schematic entry GUI.

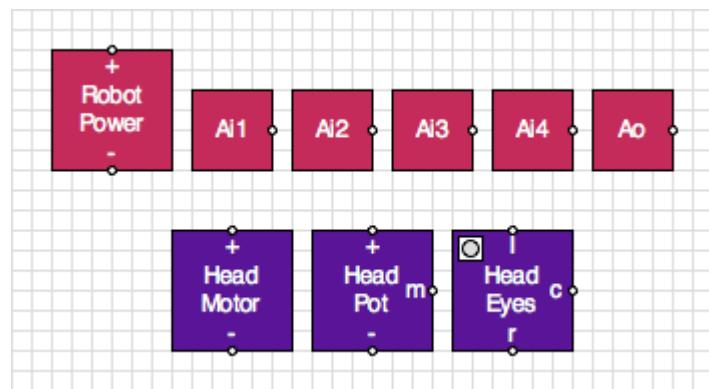


Figure A-2: Grouped components: Robot Connector and Head Connector.

A.2 Board

The board (item 1 in Figure A-1) is where the user can draw circuit schematics. The user may move a component on the board by clicking the component and dragging it to the desired place. When dragging, the GUI draws guide lines that extend to the extents of the board to help the user place the component at the right place. The user has the option to select and move multiple items at once. The user may delete a component by clicking on it while pressing `Ctrl`. The user may rotate a component by clicking on it while pressing `Shift`. An important aspect of circuit schematics is interconnecting components with wires. Each circuit component in the GUI comes with a few connection points. The user may connect components by drawing wires that connect these connection points. The user may draw a wire by clicking on a connection point and dragging. A wire may be drawn to (1) another connection point, or (2) a wire already on the board, which snaps the new wire onto the existing wire, or (3) an empty location on the board, which would create a new connection point. A wire may also be drawn starting from an existing wire, which creates a new connection point on the existing wire. The GUI allows the user to drag connection points. To achieve this, the GUI has two possible states for the cursor, the drawing states and the dragging state (mainly referring to wires). Item 2 in Figure A-1 displays the panel that lets the user toggle between these two states. In the dragging state, the user can drag connection points just like other circuit components. When drawing wires, the GUI attempts to route the wires in a way that is aesthetically pleasing. That is, the wire is routed so as to avoid crossing wires and, more importantly, wires crossing components on the board. Note that doing this is not a trivial task. In fact, this problem is very similar to the layout problem that this project aims to tackle. The solution to the wiring problem in the GUI also uses search.

A.3 Analysis

The analysis section of the GUI (item 4 in Figure A-1) lets the user analyze the drawn circuit schematic in two ways.

A.3.1 Simulation

The GUI lets the user simulate the circuit and test whether it behaves as expected. The simulation infrastructure is ported from CMax, and hence circuits are simulated exactly as they are simulated by CMax. If there are probes in the circuit, the simulator presents the voltage difference across the probes as an output. If there is a motor in the circuit, the simulator presents the motor's angle and motor's rotational velocity as functions of time. If there are any pots in the circuit, the user is expected to select a simulation file for each pot describing how the pot is manipulated as a function of time. Similarly, if the photosensors are a part of the circuit, the user is expected to select a simulation file for each photosensor set describing the lamps distance and angle from the head. These simulations help students (and staff) verify that they have a correctly functioning circuit before building it.

A.3.2 Layout

The GUI also lets the user generate a layout for the circuit schematic, which is the main object of this project. Very importantly, the GUI makes it very easy to relate the schematic with the layout. When the user hovers over a component in either window, the GUI highlights the corresponding component in the other window. When the user hovers over a wire in one window, the GUI highlights all of the wires in both windows that correspond to the same node. Figures A-3 and A-4 demonstrate this feature.

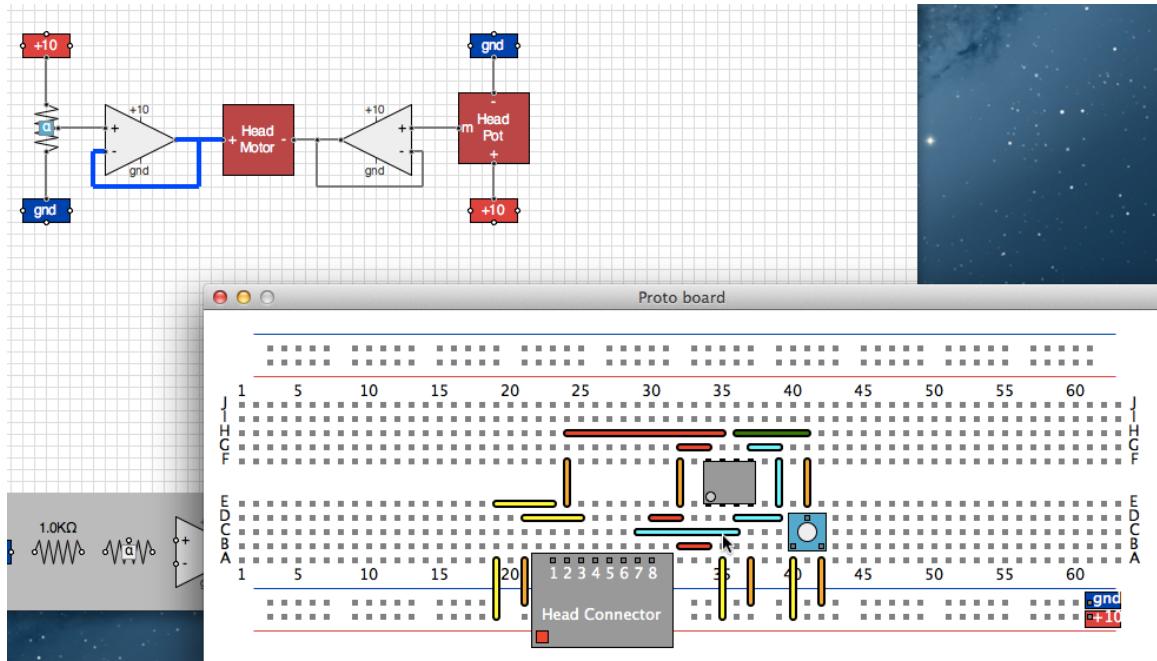


Figure A-3: Wire highlighting example.

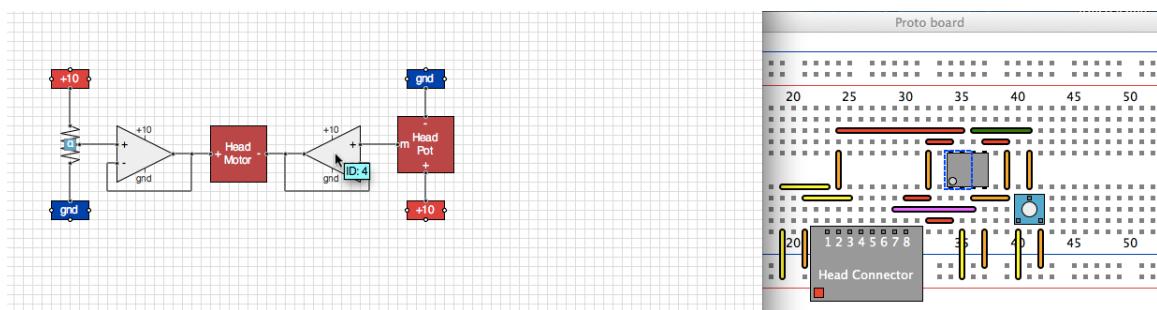


Figure A-4: Component highlighting example.

A.4 Other Features

Here we discuss several features offered by the GUI that have not been discussed so far:

1. The GUI allows the user to save schematics for later viewing or editing.
2. Protoboard layouts can also be saved as CMax files allowing for editing in CMax.
3. The schematic editing tool allows the user to undo and redo all actions.
4. The GUI has menu items that offer access to some of the features already discussed. Menu items are added when particular circuit components are selected. For instance, selecting a pot component results in a new menu item that allows the user to select a signal file for the pot. The same properties can also be reached by right-clicking on the components.
5. The GUI changes the cursor appropriately to provide feedback. For instance, the cursor becomes a pencil if the user can draw a wire starting at the cursor's current position. The cursor also changes to indicate when the user is about to rotate or delete a component. If the tool is busy either running a simulation or generating a layout, the cursor shows that the tool is busy.

A.5 Shortcuts

Table A.5 presents the shortcuts available in the GUI.

Action	Shortcut
+	New file
+	Open file
+	Save file
+	Quit
+	Undo
+	Redo
+	Close simulation windows
	Generate layout
	Run simulation
	Delete selected items
	Rotate selected item
	Toggle cursor state
	Move selected items left
	Move selected items down
	Move selected items up
	Move selected items right

Table A.1: Shortcuts.

Bibliography

- [1] Introduction to EECS I. <http://web.mit.edu/6.01>.
- [2] 6.01 Staff. Cmax.