Automatic Protoboard Layout from Circuit Schematics

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

Author . . . .
Department of Electrical Engineering and Computer Science
December 8, 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 from Circuit Schematics

by

Michael Mekonnen

Submitted to the Department of Electrical Engineering and Computer Science
on December 8, 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 build, analyze, and save circuit schematics through a graphical user interface and automatically generates protoboard layouts that are almost always easy to build and debug. 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 tool 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 Circuit Components 12

2.1.2 Circuit Schematic 13

2.1.3 Protoboard 15

2.1.4 Protoboard Layout 15

2.2 Previous Work 17

2.2.1 CMax 18

2.2.2 Current Work in Automatic Layout 20

3 Methods **21**

3.1 GUI 21

3.2 Solving the Layout Problem 21

3.2.1 Part 1: Placement 23

3.2.2 Part 2: Wiring 29

3.2.3 Combining the Methods 34

3.2.4 Evaluation 36

4 Results **41**

4.1 Random Layout 45
4.2 Comparing Placement Methods 46
4.3 Comparing Wiring Methods 52
4.4 Comparing Search Methods 61
4.5 Combined Algorithm 66

5 Discussion **73**

5.1 Search Space Size 73
5.2 Justifying Placement Choices 74
5.3 Explaining the Results 75
5.3.1 Comparing Placement Methods 75
5.3.2 Comparing Wiring Methods 77
5.3.3 Comparing Search Methods 80
5.3.4 Combined Algorithm 81
5.4 Further Work 82
5.4.1 Treating Resistors as Wires 82
5.4.2 Building Layouts Similar to Previously Generated Layouts . . 83
5.5 Remarks 83

A Schematic Entry GUI **85**

A.1 Palette 85
A.2 Board 87
A.3 Analysis 88
A.3.1 Simulation 88
A.3.2 Layout 88
A.4 Other Features 90
A.5 Shortcuts 90

List of Figures

2-1 6.01 robot 13
2-2 Circuit pieces 14
2-3 Sample circuit schematic 15
2-4 Protoboard 16
2-5 Sample protoboard layout 16
2-6 CMax 18

3-1 Schematic entry example 22
3-2 Acceptable circuit piece placements 23
3-3 Placement examples 25
3-4 Placement cost function examples 28
3-5 Random schematic generation bases 38
3-6 Sample randomly generated schematic 39

4-1 Algorithm alternatives summary 41
4-2 Schematic complexity histogram 43
4-3 Exemplar schematic 44
4-4 Random layout exemplar 45
4-5 Blocking method exemplar 46
4-6 Distance method exemplar 46
4-7 Placement method success rate comparison 47
4-8 Placement method success rate trend comparison 48
4-9 Placement method wiring time trend comparison 49
4-10 Placement method layout quality trend comparison 50

4-11 Placement method layout badness trend comparison 51
4-12 All pairs method exemplar 52
4-13 Per-node (increasing) method exemplar 52
4-14 Per-node (decreasing) method exemplar 53
4-15 Per-pair (increasing) method exemplar 53
4-16 Per-pair (decreasing) method exemplar 54
4-17 Straight wiring exemplar 54
4-18 Wiring method success rate comparison 55
4-19 Wiring method success rate trend comparison 56
4-20 Wiring method wiring time trend comparison 57
4-21 Wiring method layout quality trend comparison 58
4-22 Wiring method layout quality trend comparison (without Straight wiring) 59
4-23 Wiring method layout badness trend comparison 60
4-24 $A*$ Search exemplar 61
4-25 Best First Search exemplar 61
4-26 Search method success rate comparison 62
4-27 Search method success rate trend comparison 63
4-28 Search method wiring time trend comparison 64
4-29 Search method layout quality trend comparison 65
4-30 Search method layout badness trend comparison 66
4-31 Combined algorithm exemplar 67
4-32 Combined algorithm success summary 68
4-33 Combined algorithm success trend 69
4-34 Combined algorithm time trend 70
4-35 Combined algorithm layout quality trend 71
4-36 Combined algorithm layout badness trend 72

5-1 Expanded vertices histograms 78

A-1 Schematic entry GUI parts 86
A-2 Grouped components 86

A-3 GUI wire highlighting example 89
A-4 GUI component highlighting example 89

List of Tables

4.1 Placement method success rate comparison 47
4.2 Wiring method success rate comparison 55
4.3 Search method success rate comparison 62

5.1 Op-amp packaging possibilities 75

A.1 GUI shortcuts 91

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 easy to build, easy to debug, and aesthetically pleasing. The tool this paper discusses is geared towards circuits that students would build in the Introduction to Electrical Engineering and Computer Science I[4] course at MIT (also known as 6.01).

We are motivated to solve the layout problem in order to let 6.01 students spend more of their lab time thinking about how to design circuits and less time thinking about how to lay them out on protoboards. In this project, we used the Python programming language to develop a tool that lets students easily build and simulate circuit schematics through an intuitive graphical user interface (GUI). After building and testing a circuit schematic with the tool, a student can proceed to building the circuit on a physical protoboard based on the layout generated by the tool. Not only does the tool generate a layout, but it also displays the relationships between the original schematic drawn by the student and the generated layout. With this tool, a student's lab time will be spent mostly on designing, building, and testing circuits rather than on the difficult and arguably 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[1] generation. Chapter 3 discusses how we solved the problem, including various alternatives considered in each part of the solution, and how we evaluated 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.

[1] Throughout this paper, layout is used to mean specifically protoboard layout. In section 2.2.2, we stretch its meaning to include layout on other kinds of circuit boards, such as Printed Circuit Boards.

Chapter 2

Background

In this chapter we discuss essential background information to this project. First, we discuss the specific terminology used in this paper. Next, we discuss previous work relating to this project.

2.1 Technical Background

As this project introduces a new teaching tool for 6.01, let us first discuss the scope of circuits in 6.01.

2.1.1 Circuit Components

The rudimentary circuit components used in 6.01 are resistors, operational amplifiers (op-amps), and potentiometers (pots). In addition to these basic parts, students may build circuits to control LEGO motors or to control aspects of robots designed specifically for 6.01. One of the 6.01 robots is depicted in Figure 2-1. The robots can be equipped with heads that contain three parts held together by a shaft: a LEGO motor, a potentiometer, and a circuit card containing two photosensors. The robot in Figure 2-1 has a head attached. To connect a layout to a LEGO motor, a student would use a 6-pin connector, and to connect a layout to a robot or a robot head, a

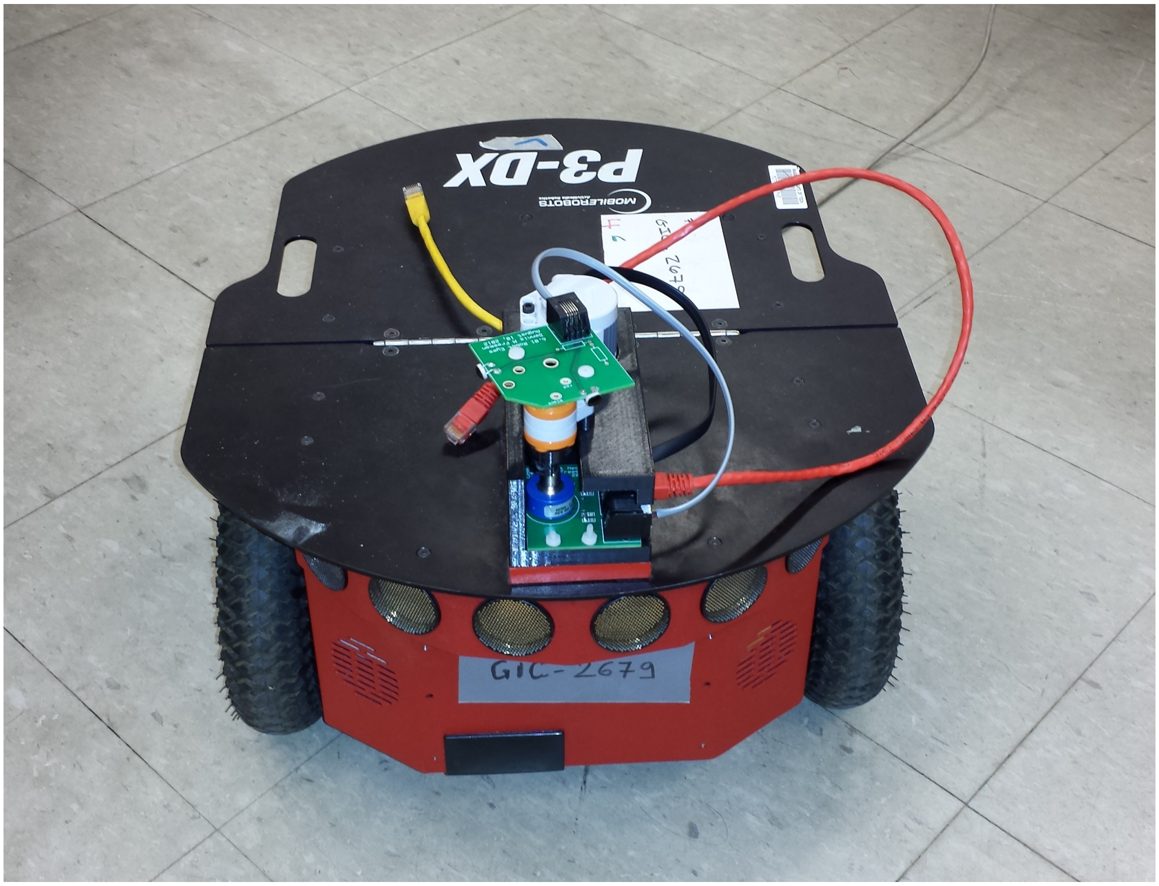
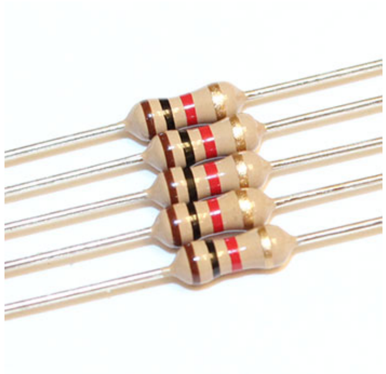


Figure 2-1: One of the 6.01 robots, with a head attached.

student would use an 8-pin connector. Figure 2-2 displays all of the circuit pieces a 6.01 student may use to build a protoboard layout.

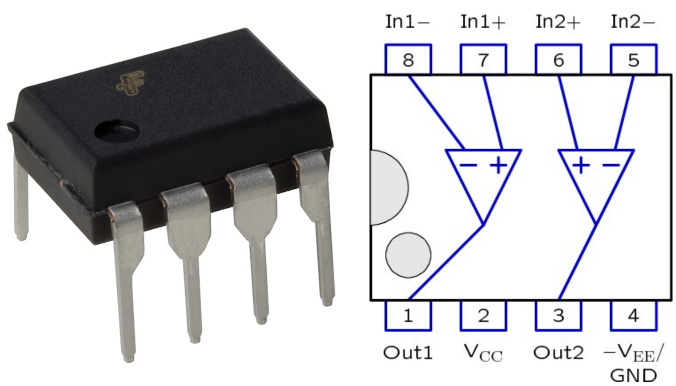
2.1.2 Circuit Schematic

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 2-3 presents an example of a circuit schematic.

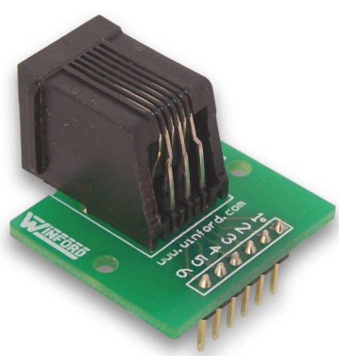


(a) Resistors

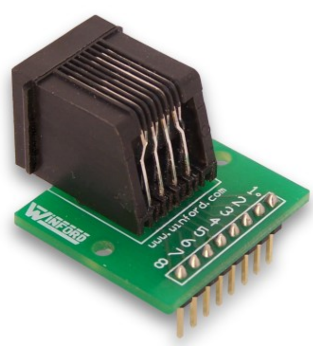
(b) Potentiometer



(c) Operational Amplifier



(d) 6-pin Motor Connector



(e) 8-pin Robot/Head Connector

Figure 2-2: All circuit pieces used in 6.01 that may be inserted into a protoboard.

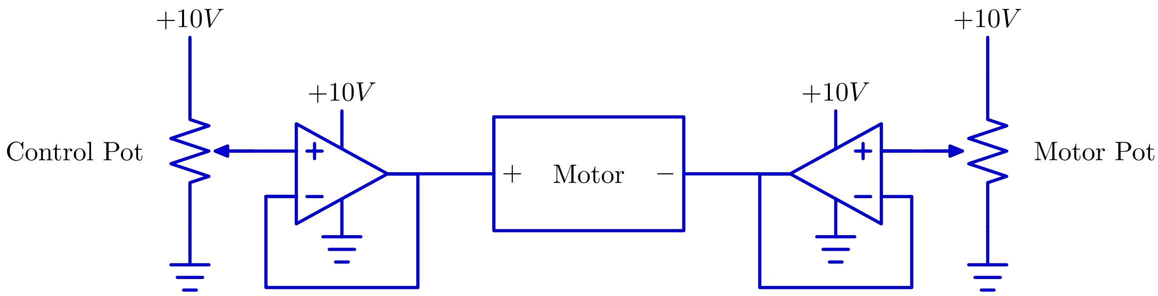


Figure 2-3: Sample schematic of a motor angular position controller circuit.

2.1.3 Protoboard

Protoboards are boards on which one can quickly build and test small circuits. They present a 2-dimensional array of interconnected dots in which circuit pieces and wires can be inserted. Figure 2-4 presents an example of an empty protoboard. In the orientation depicted in Figure 2-4, the first two rows and the last two rows of dots (each pair having one row labeled with a + and the other row labeled with a −) are internally interconnected horizontally. That is, for each of these four rows, any two dots in the same row are connected internally. These rows are often referred to as the *rails* and are often used for the power and ground nodes, the + rows being used for power, and the − rows for ground. The group of rows labeled A through E in Figure 2-4 can be better thought of as 63 columns of 5 dots. Each of these groups of 5 dots are connected internally. The same property holds for the columns within rows F through J. Henceforth, we will refer to an internally connected group of 5 dots on the protoboard as a 5-*column*.

2.1.4 Protoboard Layout

A *protoboard layout* of a given schematic is a placement of circuit pieces and wires on a protoboard that corresponds to the schematic. A protoboard layout is constructed by placing the appropriate pieces on the protoboard and then appropriately interconnecting them with wires as prescribed by the schematic. As an example, Figure 2-5 presents one possible protoboard layout of the schematic shown in Figure 2-3.

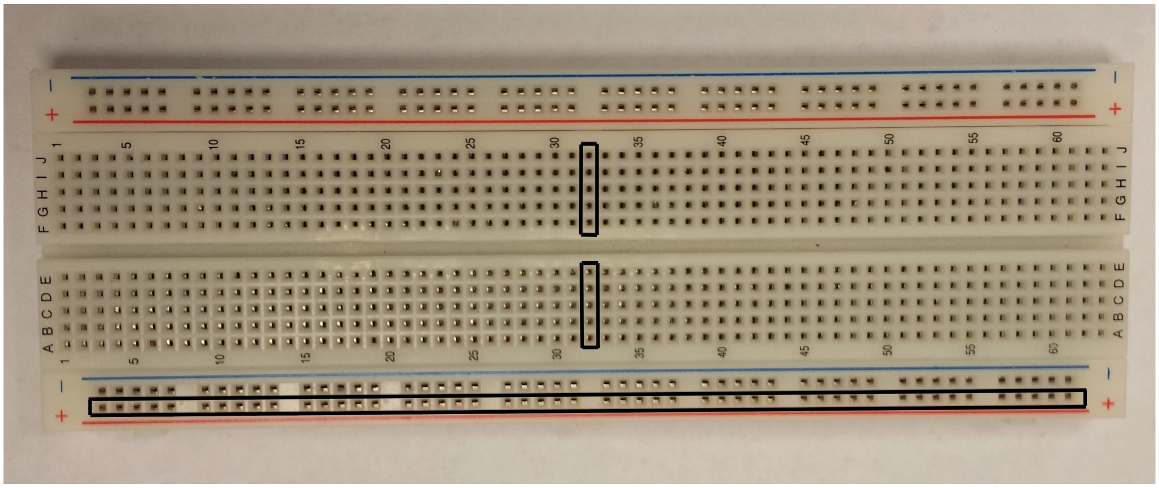


Figure 2-4: A protoboard. In the rail rows, rows labeled with + or −, the dots are internally interconnected horizontally. In the middle two groups of 5-columns, the dots are interconnected vertically.

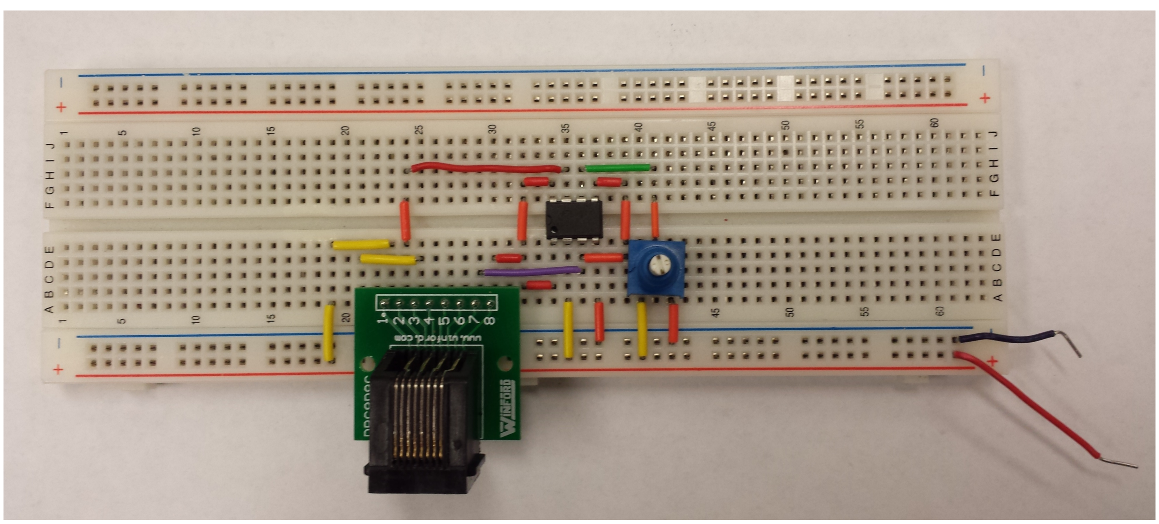


Figure 2-5: One possible protoboard layout of the schematic shown in Figure 2-3.

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, as shown in Figure 2-2(c), each op-amp circuit piece that is inserted in the protoboard actually contains two op-amp components within it. This raises an important design problem when laying out a schematic – the need to find a good way to group together the op-amps in a schematic to result in a "good" layout. To find the right packaging of op-amps and, furthermore, produce a good layout all together, we need to first understand what makes a layout good. There are no conclusive ways to tell whether a layout is good, but, keeping in mind that we want layouts that are easy to build, easy to debug, and aesthetically pleasing, we could come up with the following rules of thumb:

- The layout should not have any wires that cross circuit pieces.
- The layout should have no crossing wires, especially occlusions (i.e., crossing wires with the same orientation).
- The layout should consist of only 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 this project is automatically generating a "good" protoboard layout from a circuit schematic.

2.2 Previous Work

Here we 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 infrastructure currently used in 6.01. Next, we look at what work has been done relating to layout in general.

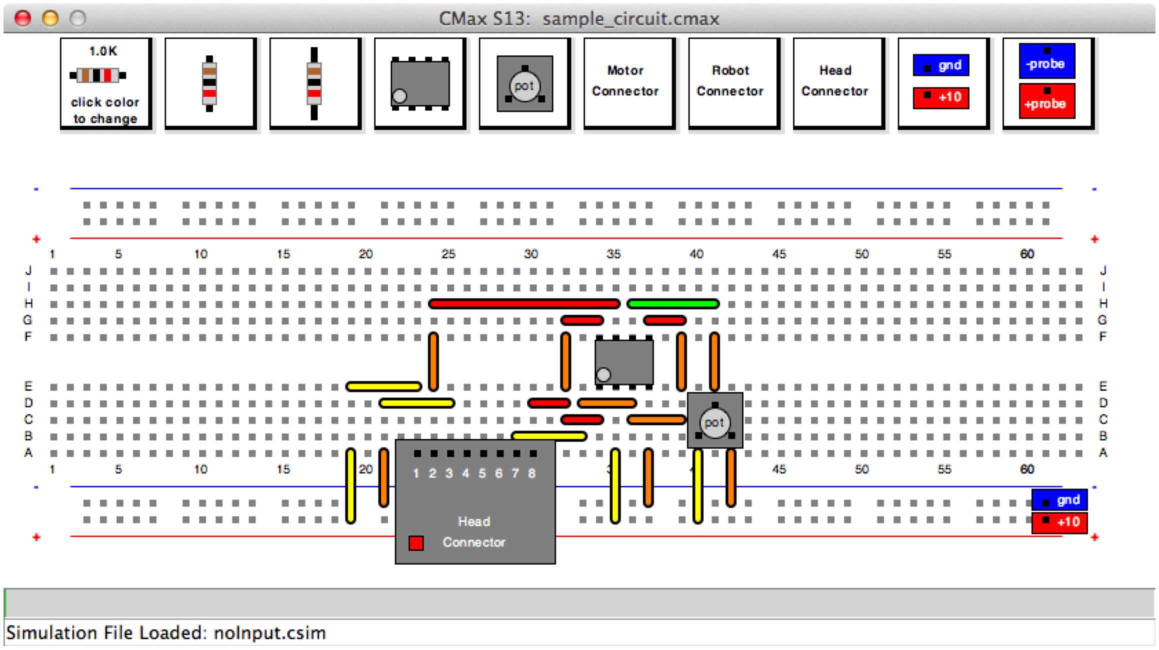


Figure 2-6: One possible CMax layout for the schematic shown in Figure 2-3.

2.2.1 CMax

In a typical circuits lab in 6.01, students first design their circuits by drawing schematics of the circuits on paper. After iteratively improving their designs based on discussions with staff members, they lay out their circuits on a simulation tool called Circuits Maximus (CMax)[5]. Note, therefore, that students currently lay out their circuits themselves. With CMax, a student can lay out a circuit on a simulated protoboard, and test the circuit to make sure that it behaves as desired. CMax provides a fast and safe way of debugging circuit layouts compared to debugging layouts on a physical protoboard. Once the students are satisfied with their observations from CMax, they build their circuits on physical protoboards and carry out the appropriate experiments. Figure 2-6 presents one possible protoboard layout, as depicted in CMax, of the schematic shown in Figure 2-3.

Using CMax has reduced circuit debugging time 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 more manageable, it provides students with a way to build, analyze, and experiment with circuits at their own leisure outside of lab.

A potential weakness of circuits labs in 6.01 as they are currently given is that student have to produces the protoboard layouts themselves. While generating protoboard layouts of circuit schematics may have instructive substance, a student's time is better spent thinking about designing circuits in the first place. Currently students design circuits by drawing schematic diagrams on paper. Once they are happy with their schematic diagrams, they proceed to laying out the corresponding circuits with CMax. When the circuits get complicated and involve many pieces, translating a schematic diagram into a protoboard layout becomes quite challenging and time-consuming. In these situations, students often end up with convoluted layouts that are difficult to debug if the circuit does not behave as expected. Not only are such layouts difficult for the students to debug, but they are also often difficult for staff members to understand. In the best case scenario, students should have to work out the right schematic diagram for the circuit they are designing, but should not have to produce the corresponding protoboard layouts.

With the schematic entry tool this paper introduces, a typical 6.01 circuits lab would proceed as follows. First, as before, students would draw schematic diagrams of their circuits on paper. Once they have schematic drawings they are happy with, they would recreate their schematic drawings on the schematic entry tool. In fact, students may proceed directly to building the schematic drawings on the tool, bypassing the experimentation on paper. Once they have a schematic drawn, they would analyze it with the tool, discuss it with staff members, and improve it with the tool. When they are satisfied with the behaviors of their schematic circuit, they would produce the corresponding protoboard layout automatically. The automatic generation of protoboard layouts would be the most important advantage of this tool. They would then build the layout on a physical protoboard and carry out experiments with it.

Avoiding the tedium of protoboard layout generation is not the only advantage of the schematic entry tool. With it, we can make circuit schematics the only mode of communication between students and staff members. Outside of lab, we communicate about circuits almost entirely by using circuit schematics. For instance, looking at the lecture and course notes given in the Spring, 2011 version of 6.01[3], we observe that almost all references to circuits are given with schematic drawings. This is because schematic drawings of circuits are particularly easy to understand. In the current 6.01 labs, communication about circuits is done by using both circuit schematics (in the experimentation stages) and protoboard layouts (in the building and testing stages). Using protoboard layouts as a way for students to describe their circuits is suboptimal because layouts require much more time and attention to understand than do schematics. The schematic entry tool has the advantage of making circuits schematics the only mode of exchanging ideas about circuits in 6.01 labs. With the tool, there will be much less need for a student or staff member to try to understand the details of a layout.

Additionally, students can keep a permanent copy of their schematics for future reference. Students currently draw schematics on paper, and often lose track of their drawing. Some students draw multiple schematics for a task, and forget which schematic they ended up using. Others misplace their schematic drawings. With the schematic entry tool, as long as students save their schematic drawings, they should always be able to refer back to them.

2.2.2 Current Work in Automatic Layout

In my explorations, I was not able to find any tools that automatically translate circuit schematics into protoboard layouts. However, there do exists tools, such as Cadence[1] and EAGLE[2], that perform partially- or fully-automatic Printed Circuit Board layout. To my findings, the owners of these tools have not published their algorithms. 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 chapter, we discuss our solution to the problem stated in Chapter 1, as well as various alternatives we considered along the way. First, we briefly introduce the schematic entry GUI. Next, we 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 an easy and intuitive task for students. Figure 3-1 gives a version of the schematic shown in Figure 2-3 as drawn in the schematic entry tool. Appendix A discusses the features and capabilities of the schematic entry GUI in further detail.

3.2 Solving the Layout Problem

In broad terms, we solved the layout problem by formulating it as a graph search problem. Given a schematic of a circuit, we start from an empty protoboard, and search through the space of all possible protoboard layouts to find a good protoboard layout, as described in Section 2.1.4, corresponding to the schematic at hand. Importantly, we utilize various simplifications and heuristics to prune out many states in the search space.

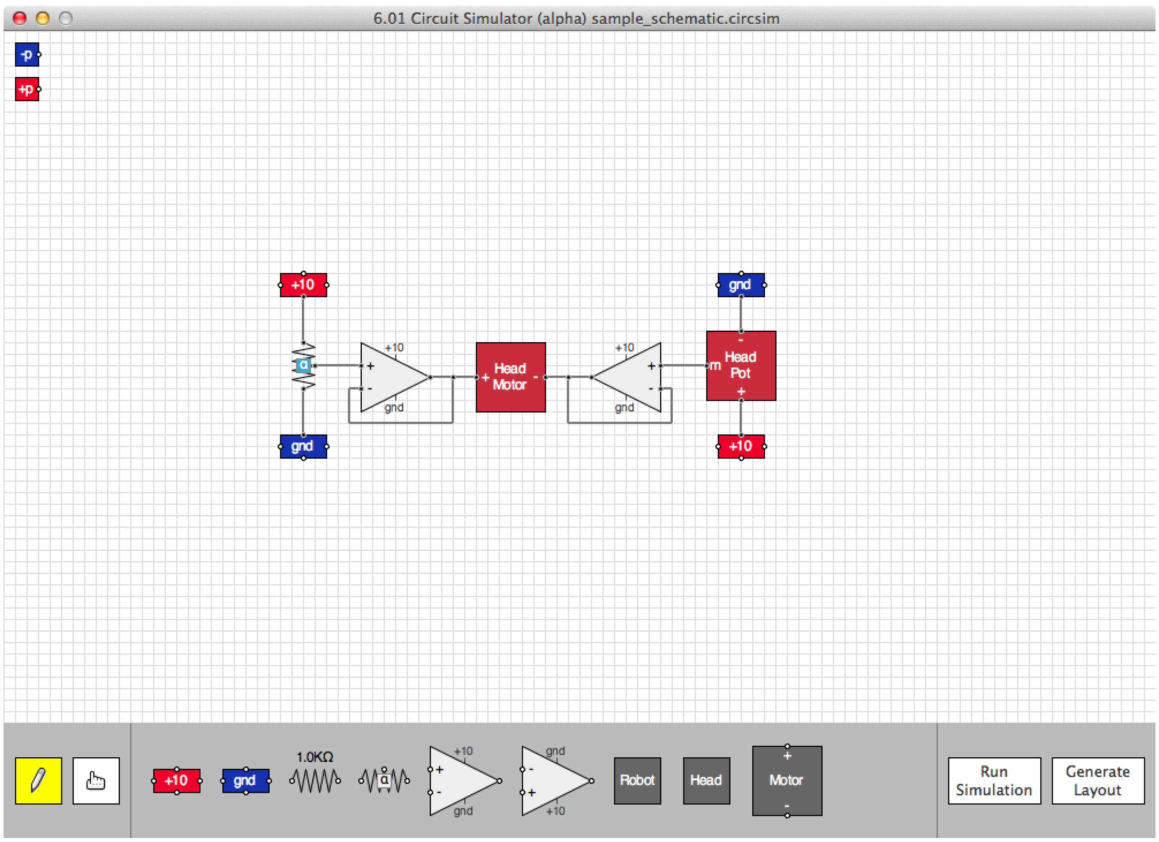


Figure 3-1: Sample schematic drawn on the schematic entry tool. This schematic describes the same circuit as the one described by the schematic shown in Figure 2-3.

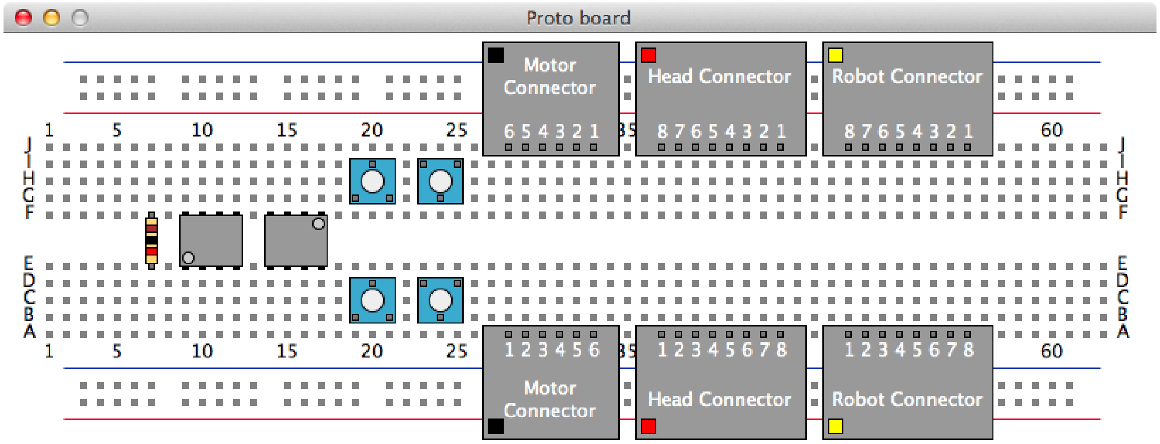


Figure 3-2: Various acceptable ways of placing each of the circuit pieces on the protoboard.

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. Pots have two possible vertical positions as well as two possible orientations. The connector pieces have two possible vertical positions each. 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 piece 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.

There are many ways of choosing a placement for a set of circuit pieces. First, we must choose from a possibly large number of ways to package together the op-amps. Section 5.2 more precisely discusses the number of different ways of packaging op-amps. For each possible packaging of the op-amps, we must consider various ways of placing the pieces on the protoboard, even with the restrictions put forth above.

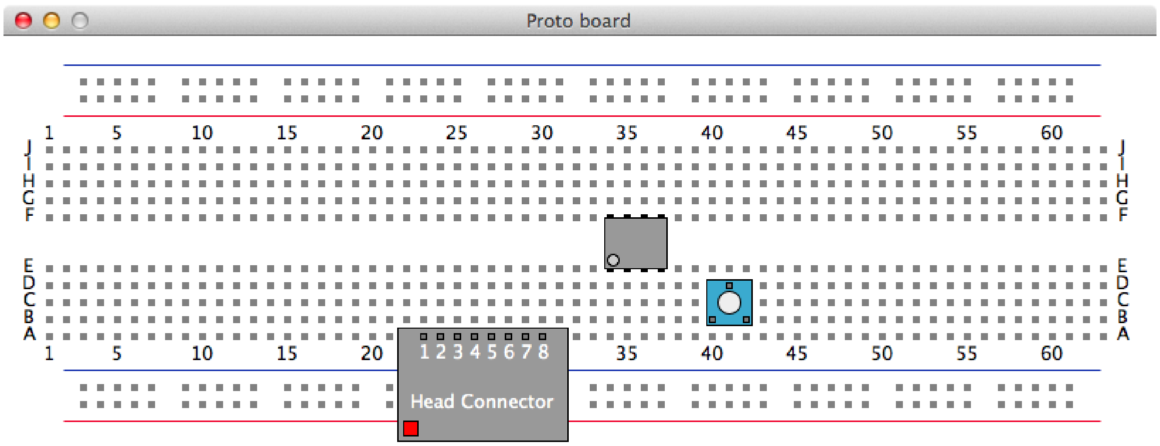
Simplifications

We reduce the number of options by only allowing placements in which no two pieces share a column. This is not necessary in general, but the number of pieces present in the most complex 6.01 circuit is small enough that any 6.01 circuit could likely be realized under this simplification.

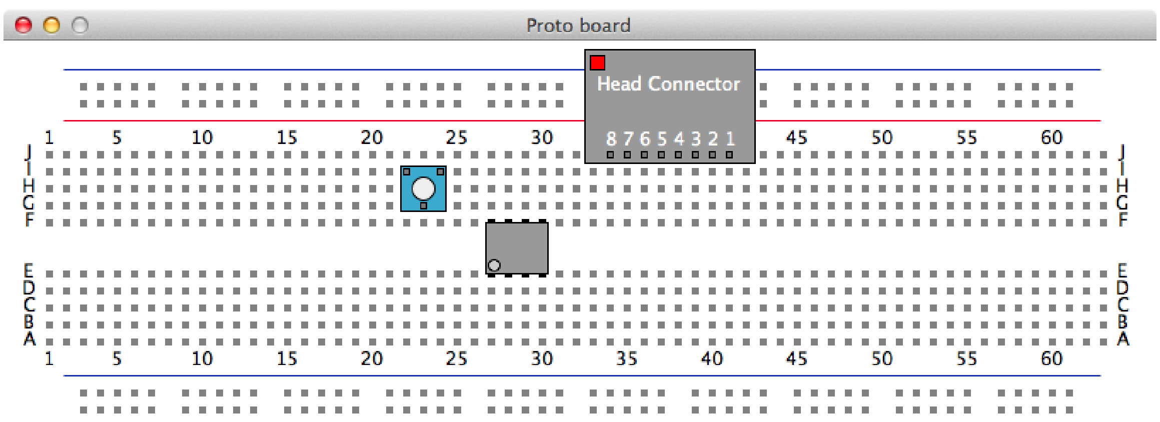
Next, we specify that there must be exactly two columns on the protoboard separating adjacent pieces, unless the pieces are both resistors, in which case there must be exactly one column separating them. These numbers of columns were chosen to leave enough space for wiring. Given a set of pieces to be placed on the protoboard, this simplification 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. Figure 3-3 shows two alternative placements for the schematic shown in Figure 3-1 that both respect the conditions put forth in the simplifications thus far. We consider a few alternatives to finding placements respecting these conditions.

Random Placement

One simple placement strategy is to choose a placement randomly. That is, to choose an op-amp packaging randomly; to choose an order of the pieces randomly; and to choose the vertical locations and orientations of the pieces randomly as well. The advantage of this approach is that it produces a placement very quickly without requiring much computation. On the other hand, it may place two pieces that need to be connected to each other very far apart, which could make the wiring task more difficult. We ought to consider alternatives that try to place the pieces so as to require as little work as possible during the wiring step.



(a) Placement 1



(b) Placement 2

Figure 3-3: Two possible placements for the schematic shown in Figure 3-1.

Small Heuristic Cost

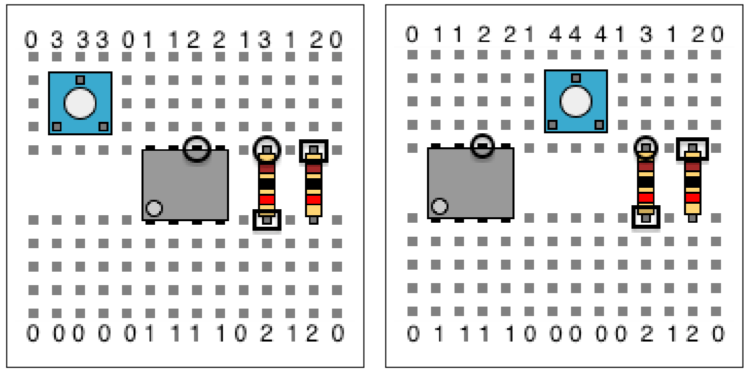
The key idea is that if two pieces are meant to be connected together by wires, then they should be placed close to each other on the protoboard. We can capture this idea by assigning heuristic costs to the placements and choosing a placement that has a small heuristic cost. To that end, there are two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

Distance Based Cost Given a circuit schematic and a corresponding placement of the circuit pieces on the protoboard, every pair of components in the schematic that is connected by wires indicates a corresponding pair of locations on the protoboard that must be connected by wires. However, we can express this requirement a little bit more concisely. We must consider all of the nodes in the schematic, and find the circuit components in the schematic that are connected to the respective nodes. For each node in the circuit, we get a set of locations on the protoboard that must be interconnected by wires. 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 cost function is the Manhattan distance between the two locations. Since we want to produce layouts that only contain horizontal and vertical wires (i.e. no diagonal wires), the Manhattan distance cost is appropriate. Given this heuristic cost for connecting two locations with wires, we can define the cost for interconnecting the locations associated with a particular node to be the weight of the minimum spanning tree of the locations. We can now 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.

We demonstrate this cost function using the two placements shown in Figure 3-4. In the Figure, each placement has two connections that must be made, the first indicated by two circled locations and the second indicated by two locations outlined by rectangles. The distance based cost for Placement 1 is $(3) + (2 + 3) = 8$ while the distance based cost for Placement 2 is $(7) + (2 + 3) = 12$. Hence, the cost function indicates that Placement 1 is a better placement of the pieces.

Blocking Based Cost The most scarce resource on the protoboard are the rows. For a given placement, we can attempt to quantify how heavily the rows will be used, and this quantity can be used as a placement cost. Given a placement, we can find a set of pairs of locations on the protoboard that need to be connected as we did above. For each 5-column on the protoboard, we can count the number of rows taken up by the piece that resides in that 5-column, if any, and the number of rows that may be taken up in connecting the pairs of locations that must be connected. This produces a cost for each 5-column that indicates how heavily the rows will be used in that 5-column. The final heuristic cost for the placement is computed as the sum of the squares of the costs for each of the $63 \times 2 = 126$ 5-columns on the protoboard. We compute the sum of the *squares* to strongly penalize heavily blocked 5-columns.

To demonstrate this cost function, let us look at the two placements in Figure 3-4 once again. Each of the 5-columns on both protoboards is labeled with its cost computed as described above. The cost for Placement 1 is the sum of the squares of the costs for each 5-column, which evaluates to 65. The cost for Placement 2, computed similarly, is 87. Once again, this cost function indicates that Placement 1 is a better placement of the pieces.



(a) Placement 1 (b) Placement 2

Figure 3-4: In both placements, there are two pairs of locations that need to be connected, denoted by either two circles or two rectangles. Using the distance based cost function, Placement 1 has a cost of 8 and Placement 2 has a cost of 12. Using the blocking based cost function, Placement 1 has a cost of 65 and Placement 2 has a cost of 87. The labels on each of the 5-columns indicate the costs for the 5-columns under the blocking cost model.

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 $10! = 3,628,800$ possible orderings. Note that this is in addition to searching over all possible ways of packaging the op-amps together. It is clear that the search for a minimal cost placement quickly gets out of hand. Rather than looking for an optimal placement, we aim for a placement with small 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.

The implementation of Algorithm 1 we use for the tool may produce different placements on multiple runs. This is a side effect of some of the data structures used to store objects – Python sets and dictionaries. As a result, the layout algorithm may generate different layouts for the same circuit on different runs.

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 ← empty Queue
R ← 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 ← Q.pop()
        Consider all vertical locations and orientations of p
        Place p at an index in R that minimizes the cost of R
        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

Once the placement task is done, the next problem is wiring. We approach this 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 6.01. Hence, students in the class may 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 and difficult.

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

[1]The 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.

[2]In 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,

vertices to consider in a priority queue, where the value associated with each vertex in 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 vertex, which is an estimate of the minimum cost to get from the vertex to a goal vertex. At each step, the algorithm pops one vertex from the priority queue (the vertex with the minimum associated value). If the vertex happens to satisfy the goal of the search, the algorithm returns the state for that vertex as the answer to the search problem. Otherwise, it adds the children vertices of that vertex to the priority queue and continues. When adding children vertices, the algorithm takes care not to reconsider states that it has 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 graph vertex will represent a protoboard layout and a set of locations on the protoboard that have yet to be connected by wires. The starting vertex will represent a partial protoboard layout that has the circuit pieces (and possibly some wires), as well as all the pairs of locations that must be connected by wires to complete the layout.

however, we will not worry about the admissibility of our heuristic as our main goal is pruning out as many states as possible, while not necessarily finding the optimal solution.

We obtain the neighbors of a vertex by taking the current protoboard layout and producing new ones in which we place exactly one new wire. We choose the starting point of the 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. For a location on a rail row, we only extend vertical wires that reach to either another rail row, or any location in either of the 5-columns that are vertically aligned. For a location on a 5-column, we extend horizontal wires that reach other 5-columns (in both directions), as well as vertical wires that reach the rail rows or any location on the other 5-column that is vertically aligned. Note that the process needs to take great care when placing new wires in order not to short, or directly connect, 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 occlusions as they are particularly difficult to physically build and debug. Finally, we want to favor making a desired interconnection between locations on the protoboard. That is, if placing one wire results in a layout in which one of the pairs of locations that need to be connected becomes connected, then the cost of that child vertex should reflect that fact. More precisely, in an attempt to connect locations loc_1 and loc_2, a wire placed extending from loc_1' to $\mathrm{loc}_{\mathrm{mid}}$, where loc_1' is a location connected (internally or by wires) to loc_1 and $\mathrm{loc}_{\mathrm{mid}}$ is a free location, the additional cost incurred by adding the wire is computed as:

$$
\begin{array}{cclcr}
\text{-100} & \times & (\mathrm{loc}_1 \text{ and } \mathrm{loc}_2 \text{ now connected}) & + & (1) \\
1 & \times & (d(\mathrm{loc}_1', \mathrm{loc}_{\mathrm{mid}}) + d(\mathrm{loc}_{\mathrm{mid}}, \mathrm{loc}_2) \text{ - } d(\mathrm{loc}_1, \mathrm{loc}_2)) & + & (2) \\
100 & \times & (\text{number of crossed wires}) & + & (3) \\
10, & & & & (4)
\end{array}
$$

where $d(\mathrm{loc}_i, \mathrm{loc}_j)$ is the Manhattan distance on the protoboard from loc_i to loc_j. Line (1) decreases the cost by 100 if a new connection is made. Line (2) penalizes long wires, taking into account how much closer (or farther) the new wire got us to connecting locations loc_1 and loc_2. Line (3) adds a cost of 100 for each new pair of crossing wires. Line (4) adds 10 to the total cost to penalize having too many wires. We produced this cost metric experimentally by thoroughly testing various ideas on a selected set of circuit schematics.

An important consideration we need to make is how 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. We considered the following six different strategies to carry out 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 represent this set of pairs of locations. In this strategy, we run exactly 1 search to solve the problem.

2. Per-node (increasing): Treat each node individually. That is, iteratively connect the locations for each of the nodes until there are no more disconnected nodes in the circuit. In this strategy, we run a number of searches equal to the number of nodes in the circuit. Order the searches in *increasing* order of the number of locations per node, breaking ties arbitrarily.

3. Per-node (decreasing): Similar to per-node (increasing), but order the searches in *decreasing* order of the number of locations per node.

4. Per-pair (increasing): Treat each pair of locations that needs to be connected individually. That is, iteratively connect pairs of locations that need to be connected until there are no more disconnected pairs. In this strategy, we run a number of searches equal to the number of pairs of locations that must be

connected. Order the searches in *increasing* order of the Manhattan distance between the pairs of locations, breaking ties arbitrarily.

5. Per-pair (decreasing): Similar to per-pair (increasing), but order the searches in *decreasing* order of Manhattan distance between the pairs of locations.

6. Straight: As a back-up alternative, we consider using one (possibly diagonal) wire to connect each of the pairs of locations that must be connected. This approach requires no search and does not take layout quality into consideration.

The strategy we choose among these six has a significant effect on the outcome of the wiring step. We will discuss the differences in detail in Chapter 4.

Goal test

We say that a given vertex is a goal vertex by verifying that its representation indicates no further pairs of locations to connect.

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 $(\mathrm{loc}_1, \mathrm{loc}_2)$ that need to be connected, we could consider the pair's distance from a 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 $A*$, vertices are considered in order of decreasing heuristic value, without consideration for the cost incurred on the path from the starting vertex.

Limiting the number of expanded vertices

With the implementation of $A*$ 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 search space size is very big (Section 5.1 discusses the search space size in more detail), 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, it gives up. We set this limit to 300 vertices. In Chapter 4 we provide data that motivates this choice and describes the effect of this choice on each of the alternatives discussed above.

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 be able to generate a layout for any schematic. 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 is discussed in Chapter 5, based on the data we obtained for the alternatives described thus far. Algorithm 2 presents the combined algorithm.

Algorithm 2 uses the per-pair wiring scheme discussed above, and 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. 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 algorithm picks one of the four unfinished layouts that has the fewest disconnected pairs of locations (breaking ties by considering wire lengths and wire-piece crossings) and completes the solution by connecting the disconnected pairs using one wire per pair chosen to maximize goodness among all equivalent pairs of locations. 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

Algorithm 2: Layout algorithm obtained by combining multiple alternatives.

Data: A circuit schematic C.
Result: A protoboard 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 unfinished layout with fewest and most compact disconnected pairs.
Connect remaining pairs with shortest possible wires (possibly diagonal).
Post-process and return resulting layout.

on the protoboard that need to be connected where all of the protoboard locations for at least one of the nodes are occupied, which is highly unlikely. This high success rate comes at the cost of placing wires that may significantly reduce the goodness of the layout.

The algorithm starts out by connecting the top and bottom rail rows of the protoboard so that all rail rows are used to connect to power and ground, and no other nodes. This is a restriction that makes it easier to debug and amend the resulting layout. Without this restriction, some rail rows might be used for nodes that are neither power nor ground, and this may confuse some students.

The algorithm also has a post-processing step that attempts to to improve the layout. The post-processing step makes three types of simple changes to the layout:

- We throw away any superfluous wires that do not serve to connect two parts of the circuit. Superfluous wires may be added to the layout in the search done by the wiring step, though very rarely.

- 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.

- If shifting a horizontal wire up or down results in a layout with fewer crossing wires, we make that change.

The last important aspect of this final algorithm not explicitly stated in Algorithm 2 is that the algorithm that will be put in front of students will only be allowed to use wires of a select few lengths. The kits that students work with 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 because they are difficult to insert and remove from physical protoboards and are also difficult to see.

3.2.4 Evaluation

Here we present how we evaluated our solution to the layout problem to test how well it would serve students in 6.01. We tested the layout tool on numerous schematics and analyzed its performance on generating layouts from those schematics. As manually generating numerous test schematics is tedious and time-consuming, we devised a method to randomly generate thousands of test schematics. As the tool targets 6.01 labs, we tried to design the randomly generated schematics so that the range of complexity of these schematics mimics the range of complexity of circuits that students may build in 6.01.

The random schematic generation goes as follows. We created 6 basic parts of schematics. These 6 bases are:

- Three resistors arranged in a T-shaped configuration.
- Two resistors in series connected to a follower op-amp configuration.
- A pot connected to a follower op-amp configuration.
- A motor.
- A robot head.

- A robot.

These bases are depicted in Figure 3-5. They cover all of the components that may be necessary in a 6.01 circuit. Each base offers at least 3 points of connection with other bases. The random generation algorithm takes all possible combinations of up to 6 bases, allowing for repetition of bases with some restrictions. The robot head and robot 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 and follower op-amp base 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 two bases, T-resistor configuration and two resistors in series together with a follower op-amp, can be repeated up to 6 times. For a given combination of bases, we generate a set of schematics in which we randomly make connections between the bases. Figure 3-6 presents a sample randomly generated schematic.

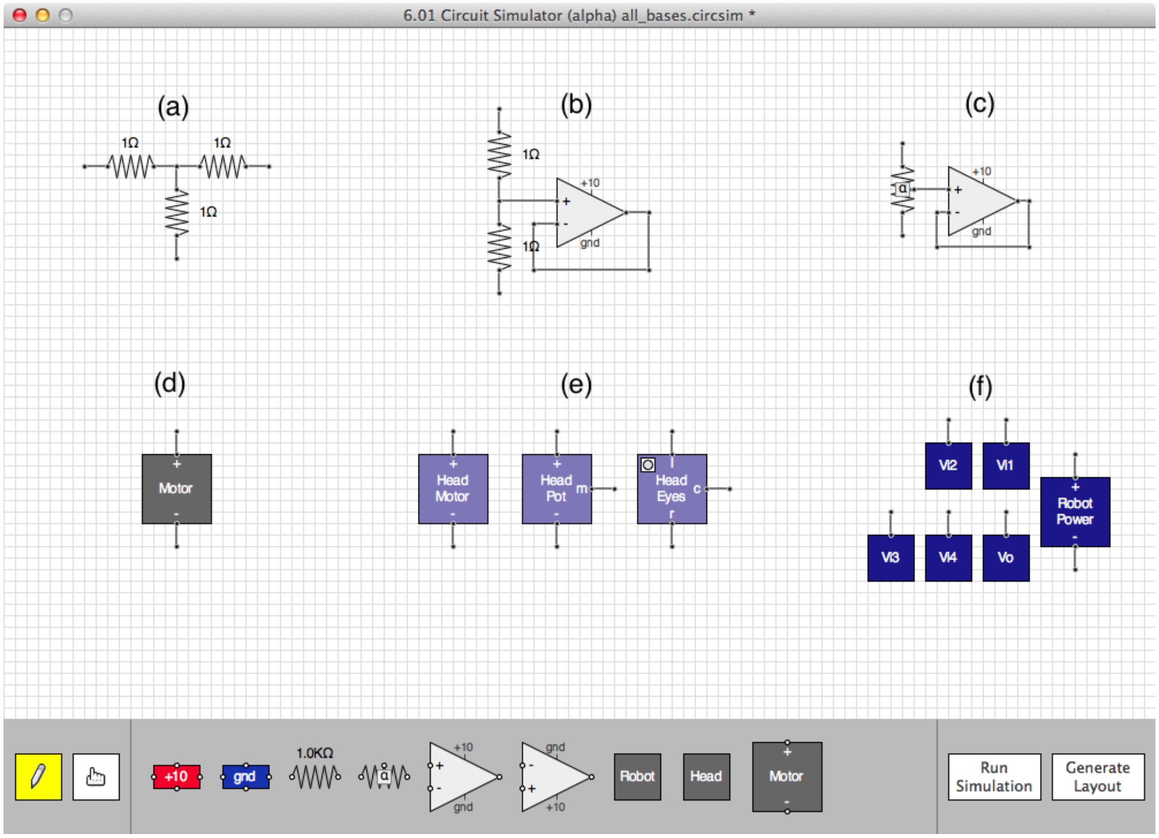


Figure 3-5: Bases for the random schematic generation scheme: (a) three resistors arranged in a T-shaped configuration; (b) two resistors in series connected to a follower op-amp configuration; (c) pot connected to a follower op-amp configuration; (d) motor; (e) robot head; and (f) robot.

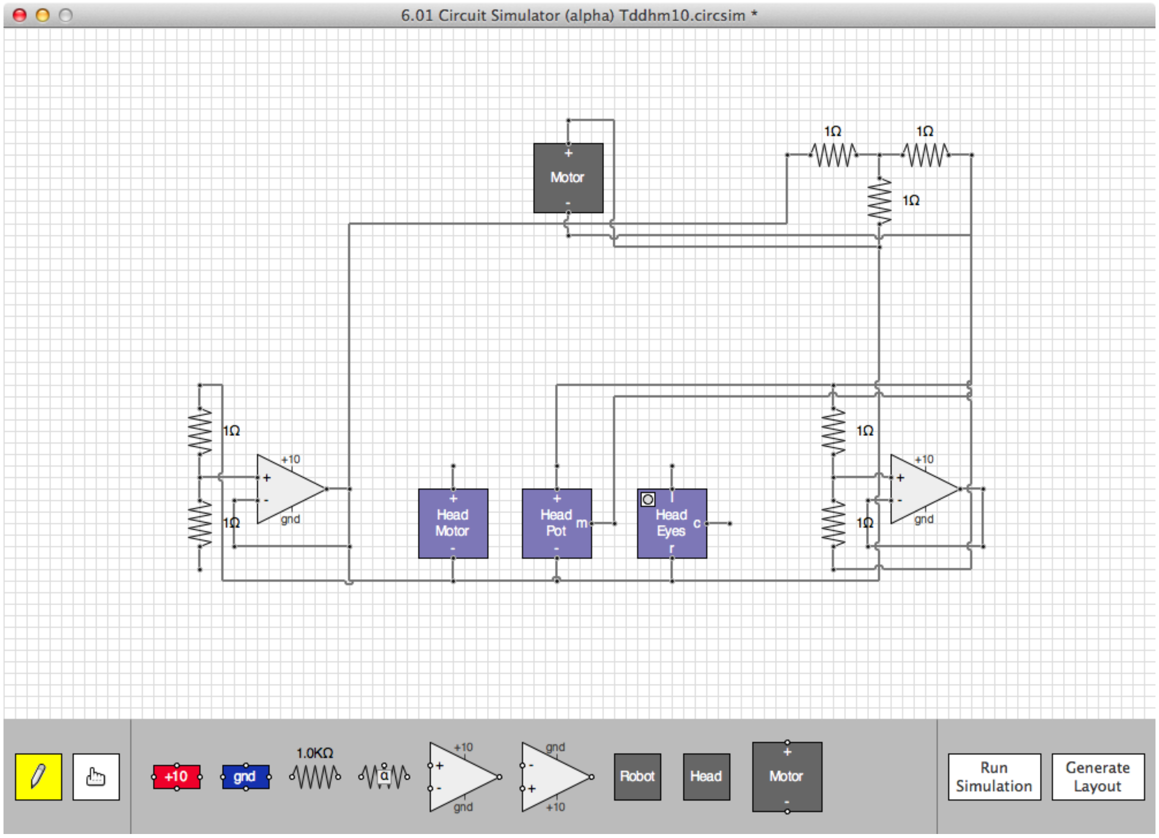


Figure 3-6: Sample randomly generated schematic.

Our scheme produces a total of 4425 test schematics. When testing a particular algorithm on these test schematics, we run the algorithm on each test schematic 10 times. Chapter 4 presents the data collected in this manner and compares 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. We define the badness of a layout to be:

$$
\begin{array}{rcll}
1 & \times & \text{(number of wires)} & + \\
2 & \times & \text{(total wire length)} & + \\
10 & \times & \text{(number of wire crosses)} & + \\
10 & \times & \text{(number of diagonal wires)} & + \\
50 & \times & \text{(number of wire-piece crossings)} & + \\
500 & \times & \text{(number of wire occlusions).} &
\end{array}
$$

We use this metric to decide which of a given set of alternative layout generation strategies tends to produce better layouts. The weights in the metric were chosen to reflect how bad each of the attributes is relative to the others. 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 a pair of wires that cross.

- Having a wire that crosses a circuit piece is about as bad as having 10 pairs of wires that cross.

- Having a wire occlusion is about as bad as having 10 wires that cross circuit pieces.

Note well that the badness metric described here is different from the cost metric used in the wiring search as described in Section 3.2.2. This badness metric is used to evaluate layouts produced by the algorithm, which may use $A*$ search in which the costs of vertices are computed, not using this badness cost metric, but the cost metric described in Section 3.2.2.

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 summarizes the alternatives. In this chapter, we provide quantitative data comparing the alternative strategies, and the data will be discussed in detail in Chapter 5.

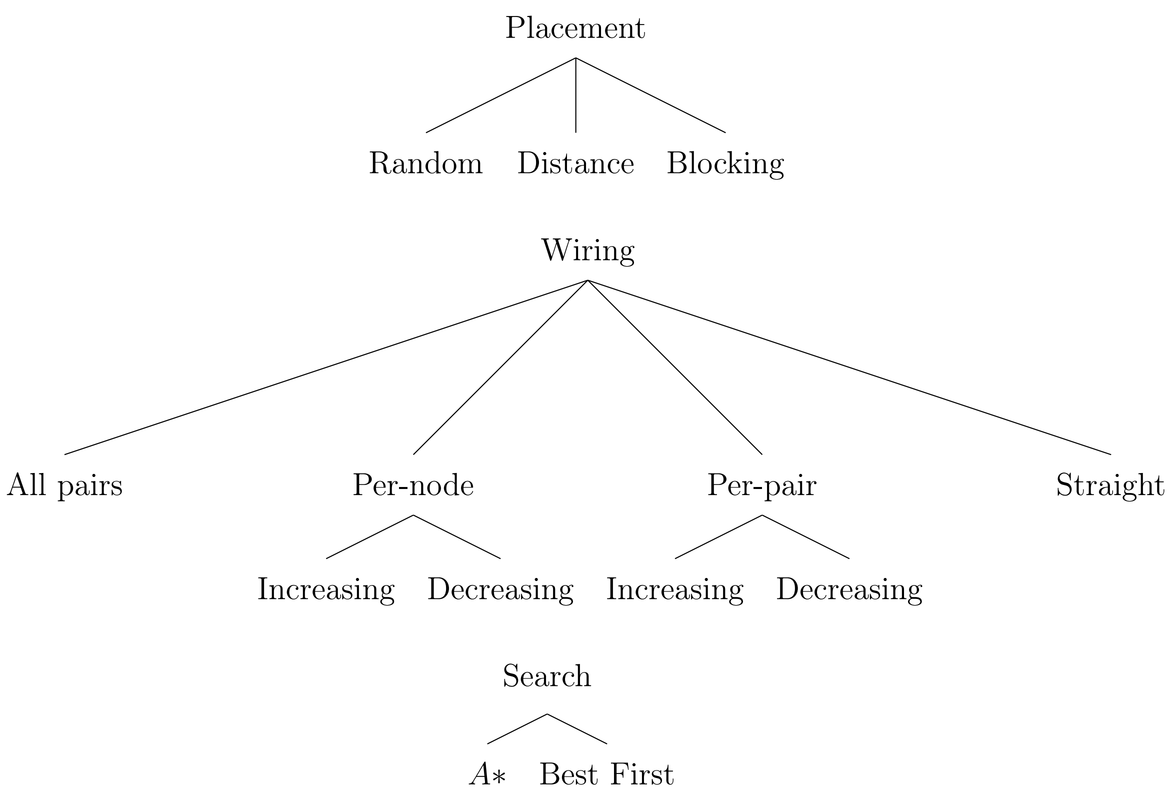


Figure 4-1: Summary of possible alternatives to the algorithm.

As comparing all $3 \times 6 \times 2 = 36$ possible implementations of the algorithm is tedious, we analyzed the three different means for alternatives (placement, wiring, and search) separately. We carried out the following comparisons:

1. Placement: Blocking vs. Distance vs. Random. The wiring method was per-pair (decreasing), and we used $A*$ Search.

2. Wiring: All pairs vs. Per-node (decreasing) vs. Per-node (increasing) vs. Per-pair (decreasing) vs. Per-pair (increasing) vs. Straight. The placement method was blocking, and we used $A*$ Search.

3. Search: $A*$ vs. Best First. The placement method was blocking, and the wiring method was per-pair (decreasing).

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

In comparing alternatives, we consider 3 questions:

1. Which alternative is successful most often?

2. Which alternative, when successful, takes the least amount of time?

3. Which alternative, when successful, produces the best layouts?

We are also interested in how each of these attributes (success rate, running time, and layout quality) varies with circuit complexity. To quantify the complexity of a circuit, we look at the number of pins in the circuit, where a pin is defined to be a connection point on a circuit component that is connected by wires to another connection point (on the same component or a different component). Figure 4-2 presents a histogram of the number of pins in the schematics in dataset that was used to do all comparisons in this chapter, not including the data presented in Section 4.5, for which we used a newly generated dataset of schematics to analyze the performance of the combined algorithm. Note that there are fewer samples of schematics of the lowest and highest circuit complexities. Hence, the statistics given for the extreme complexities are less informative.

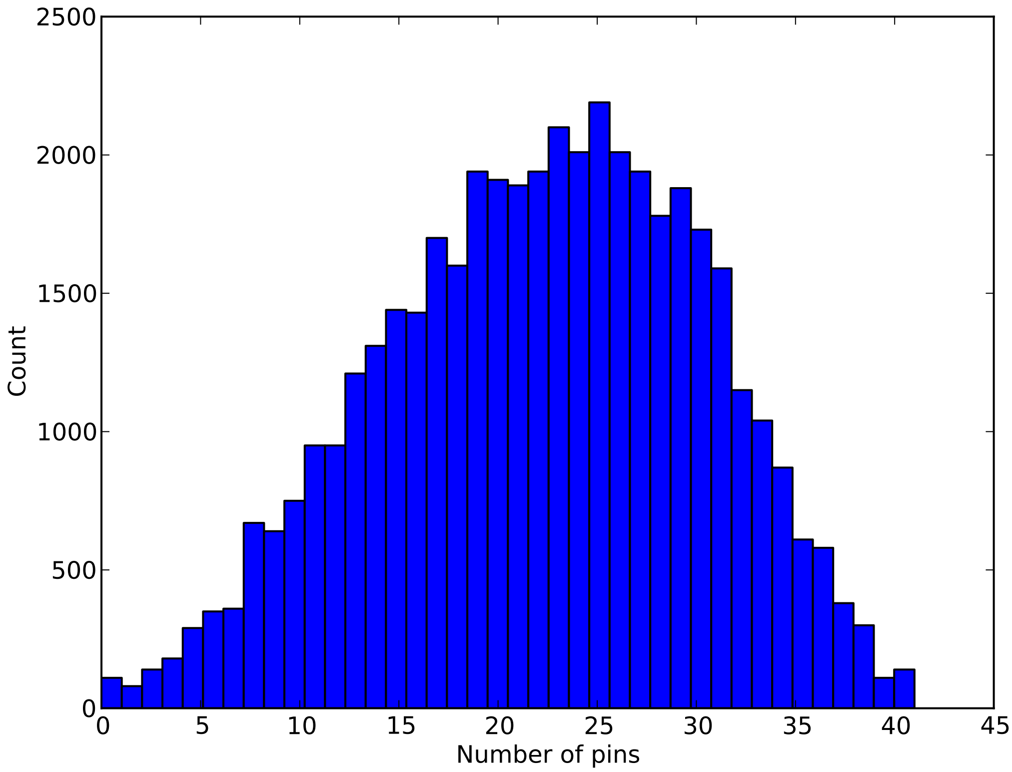


Figure 4-2: Histogram of the complexities, in terms of numbers of pins, of the 4425 schematics used for evaluation.

To compare success rates, we will look at number of successes out of 10 runs on each of the 4425 schematics. To compare running time, we will look at CPU time spent on the wiring step, as the placement step has much less variability. To compare the goodness of layouts, we will look at numbers of wires, total lengths of wires, numbers of wire crosses, and our layout badness metric as functions of circuit complexity. 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.

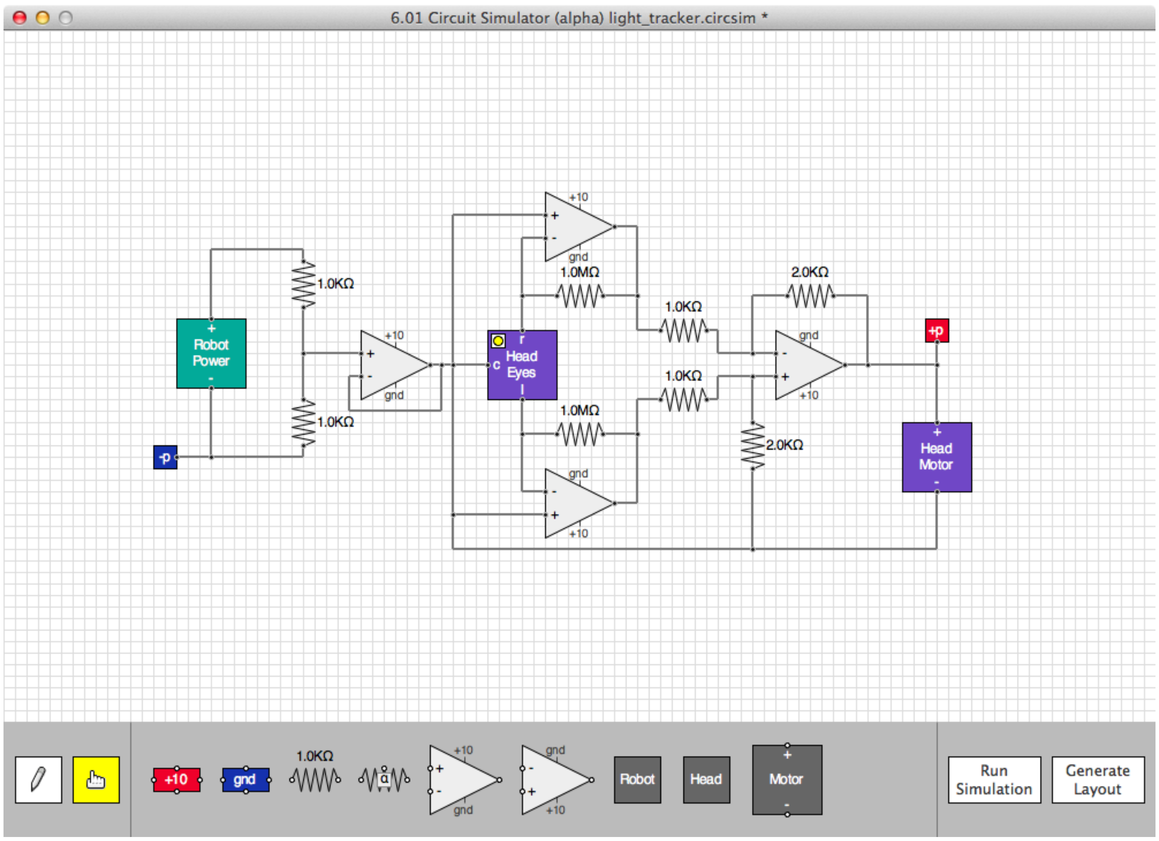


Figure 4-3: Schematic used to generate the exemplar protoboard layouts.

4.1 Random Layout

Before embarking upon the comparisons, we give an exemplar of a protoboard layout generated completely at random. Here, we choose an op-amp packaging randomly, and we place each circuit piece randomly, only taking care not to place two pieces that share a 5-column, and not obeying any other restrictions. This method of random placement, is, therefore, different from the one described in Section 3.2.1. We then use Straight wiring. Figure 4-4 presents a layout generated for the schematic shown in Figure 4-3. This completely random layout method is compared against the final algorithm in Section 4.5.

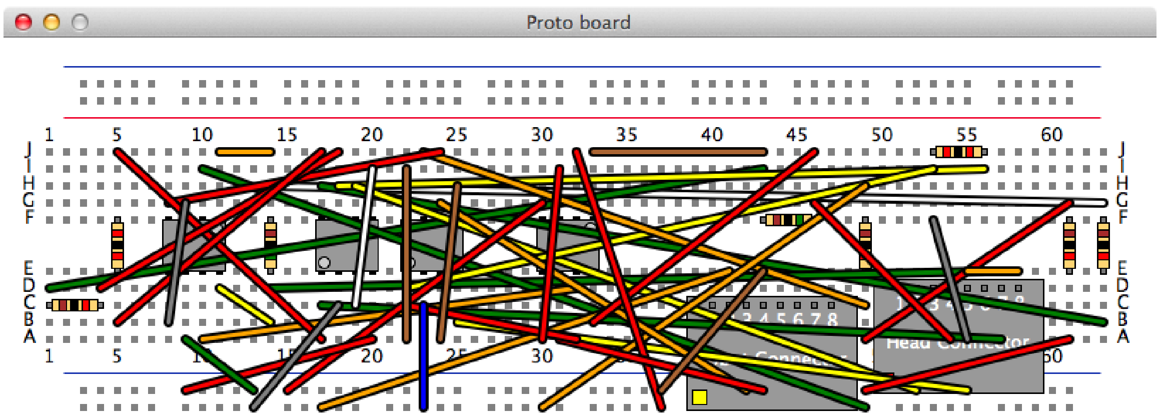


Figure 4-4: Exemplar for the completely random method.

We now look at the comparison data.

4.2 Comparing Placement Methods

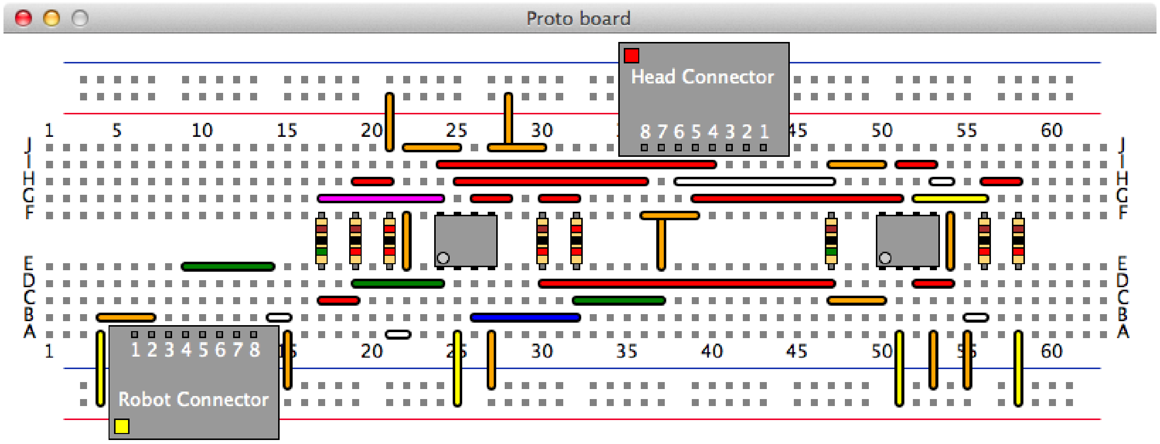


Figure 4-5: Exemplar for the blocking placement method, using per-pair (decreasing) wiring and $A*$ Search.

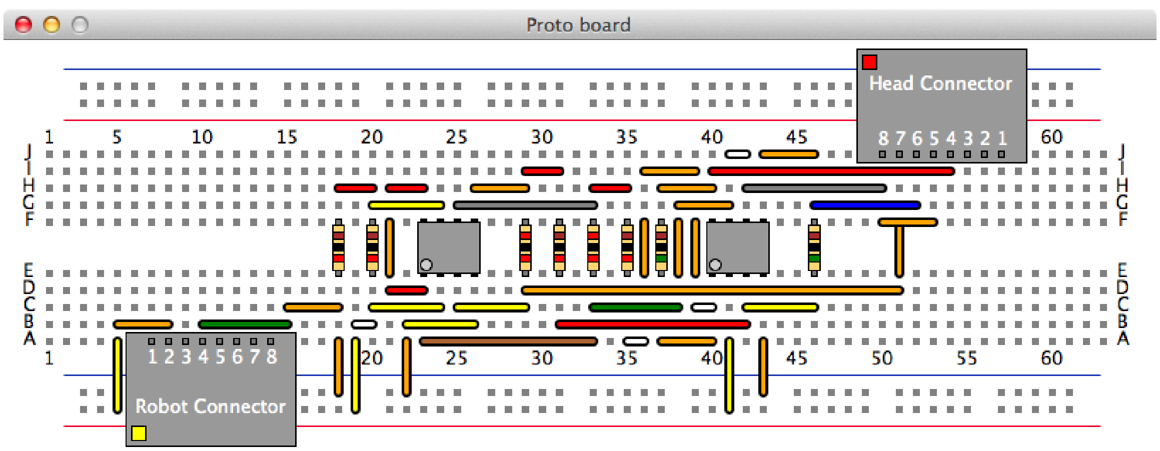


Figure 4-6: Exemplar for the distance placement method, using per-pair (decreasing) wiring and $A*$ Search.

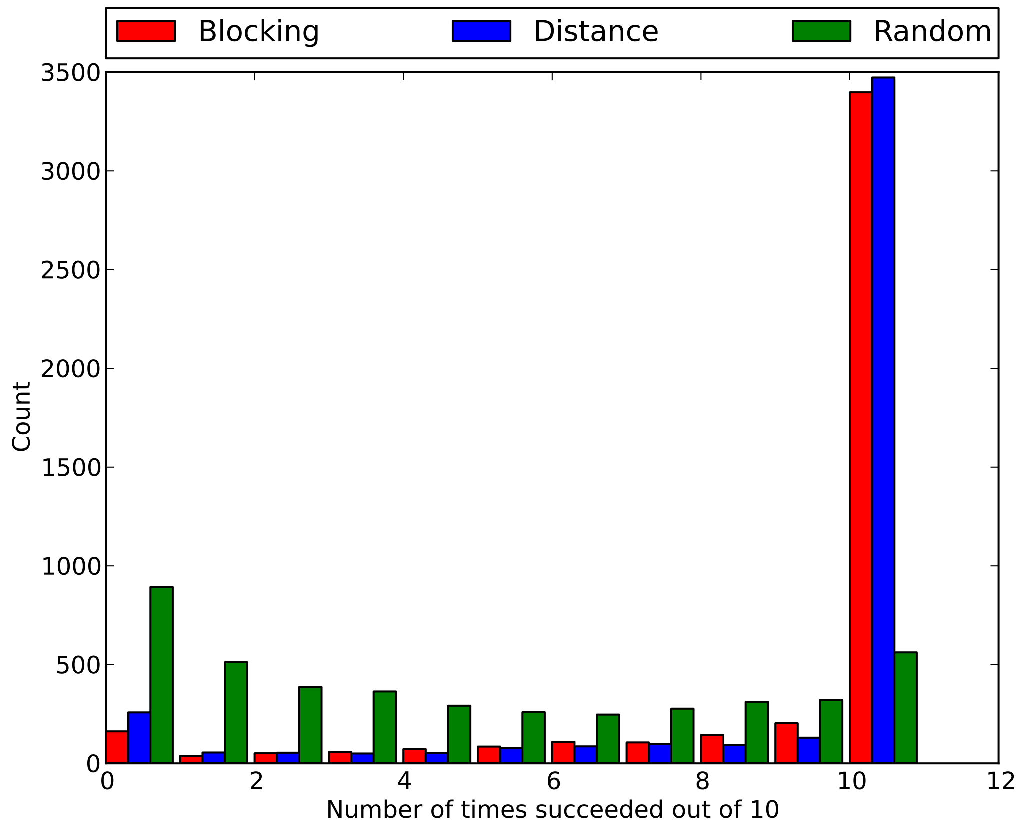


Figure 4-7: Placement method success rate comparison.

	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: This is an alternative presentation of the data given in Figure 4-7. Each cell in the table gives the count (and percentage out of the 4425) of schematics for which a particular method succeeded a given number times out of 10 runs.

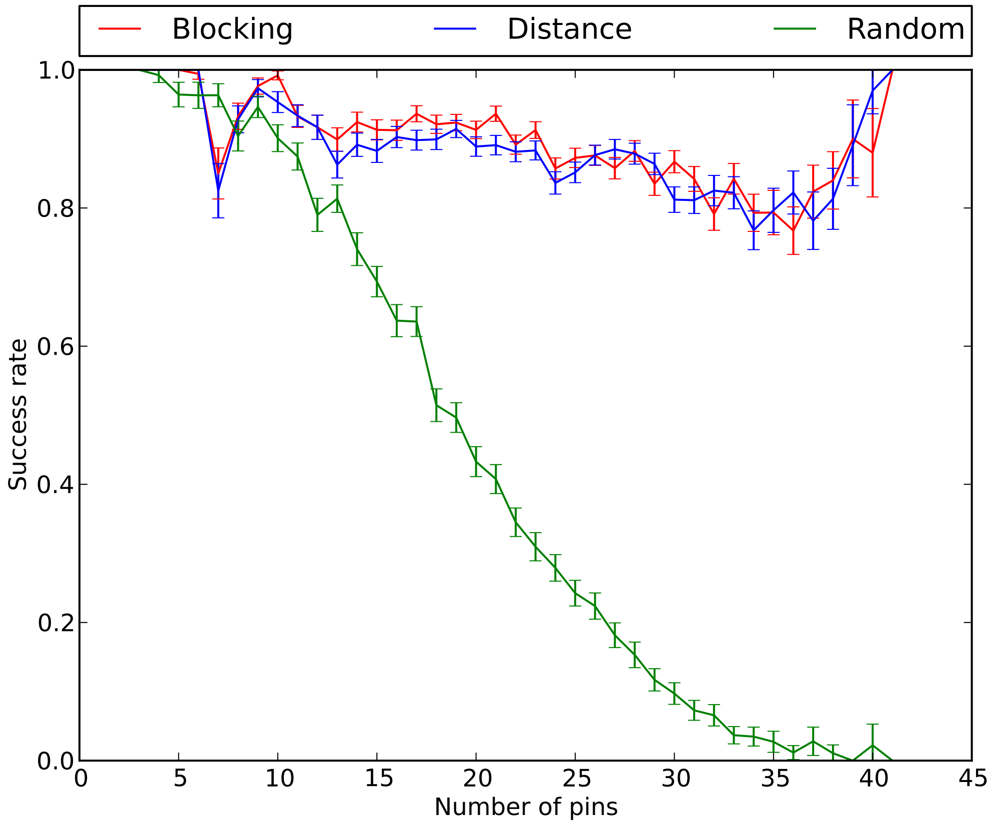


Figure 4-8: Placement method success rate trend comparison.

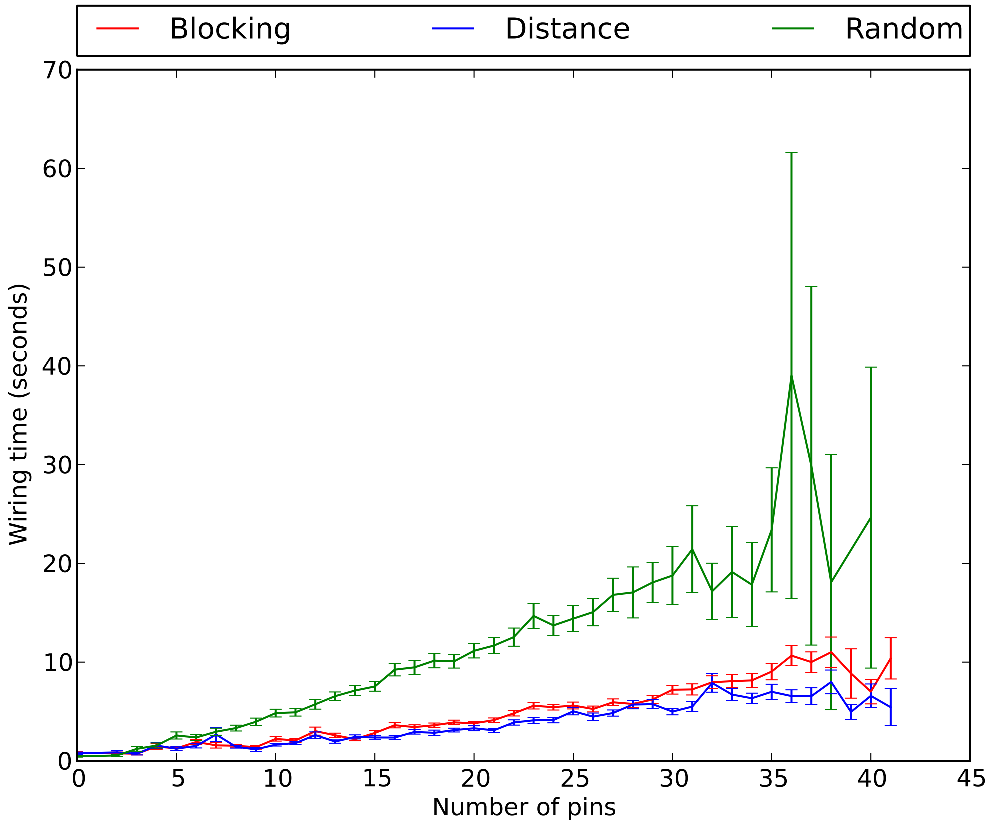


Figure 4-9: Placement method wiring time trend comparison for successful runs.

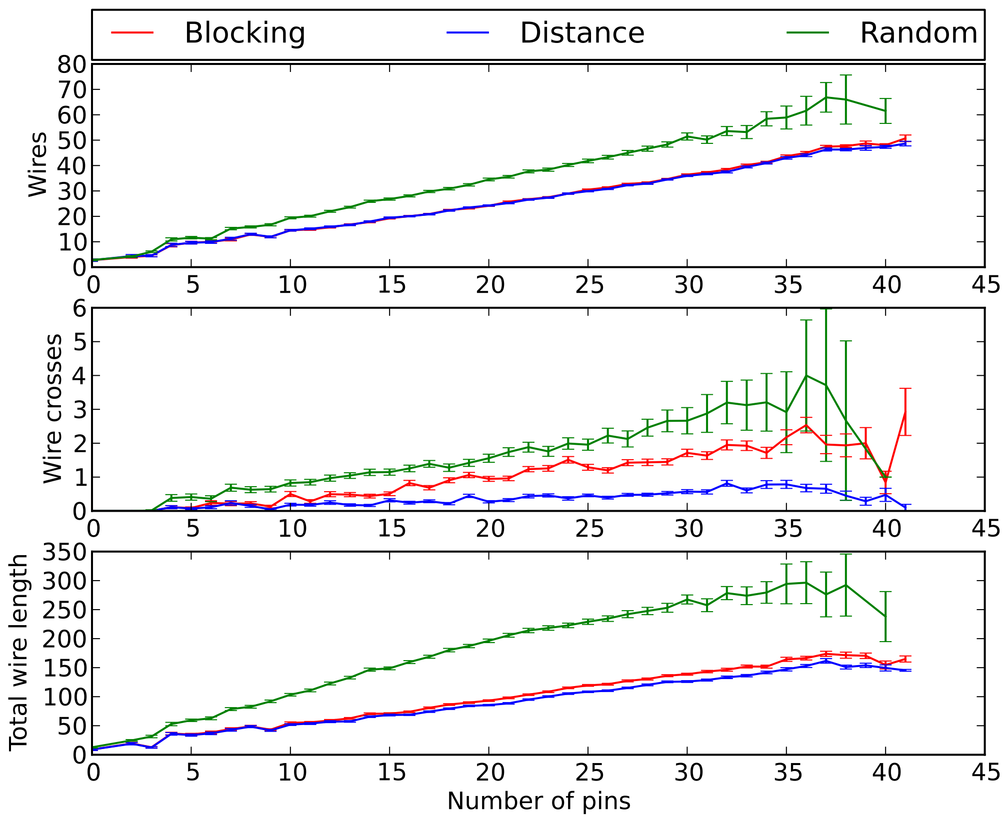


Figure 4-10: Placement method layout quality trend comparison.

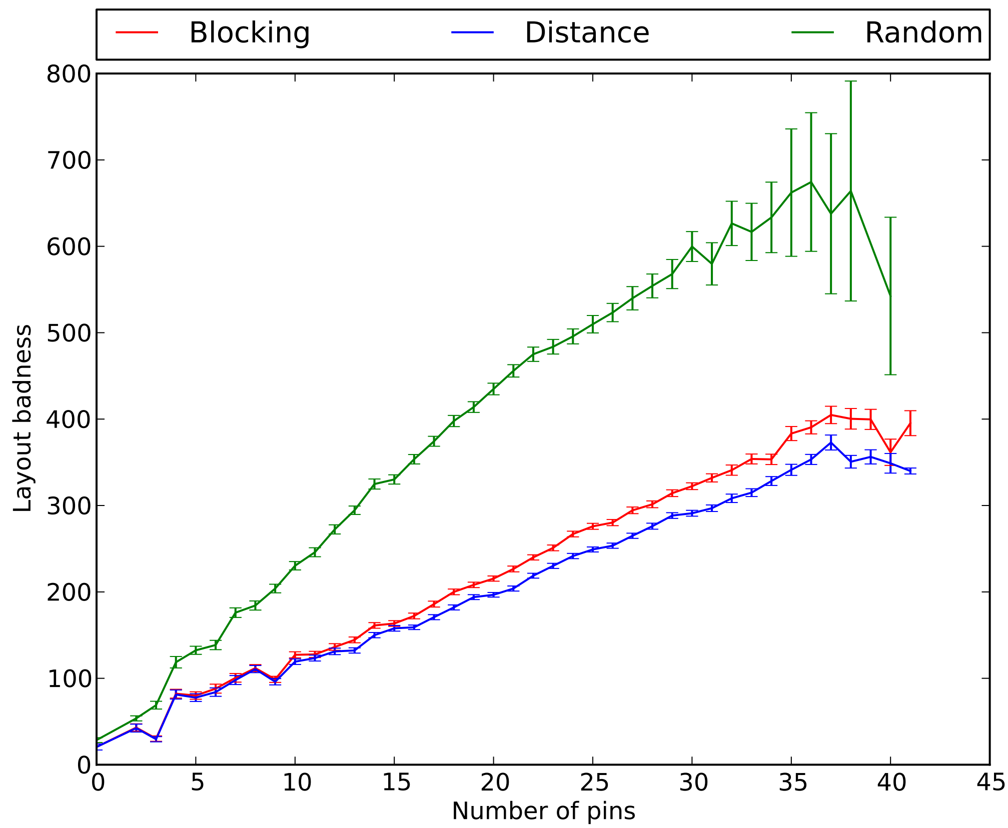


Figure 4-11: Placement method layout badness trend comparison.

4.3 Comparing Wiring Methods

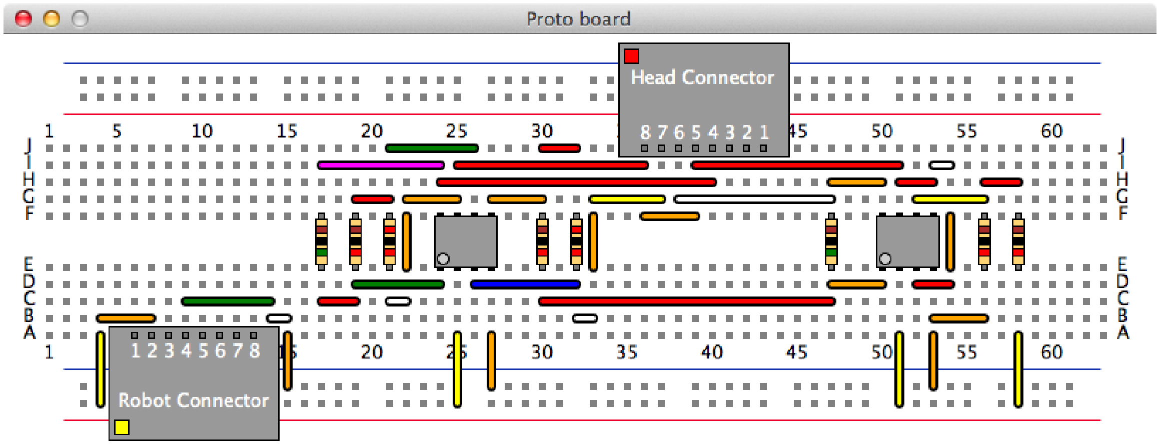


Figure 4-12: Exemplar for the all pairs wiring method, using blocking placement and $A*$ Search.

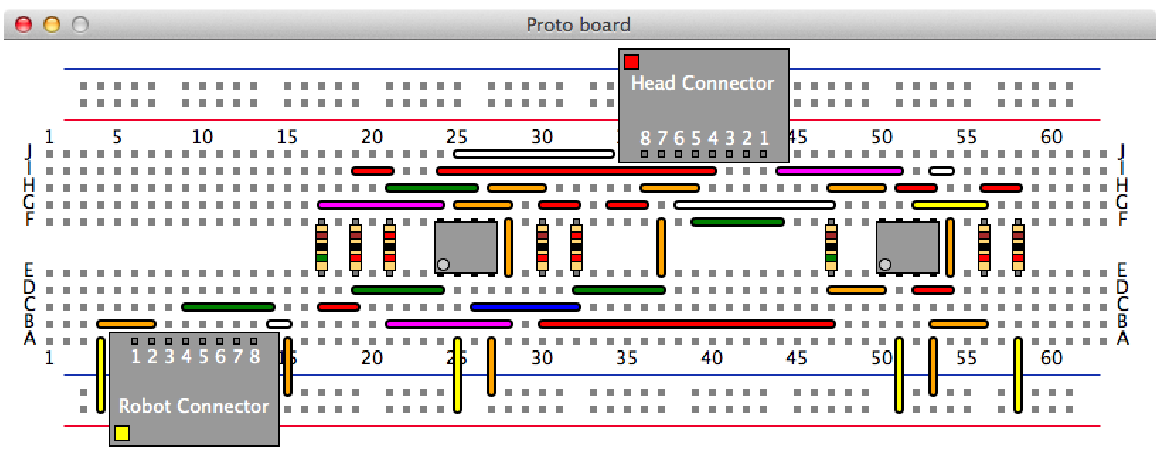


Figure 4-13: Exemplar for the per-node (increasing) wiring method, using blocking placement and $A*$ Search.

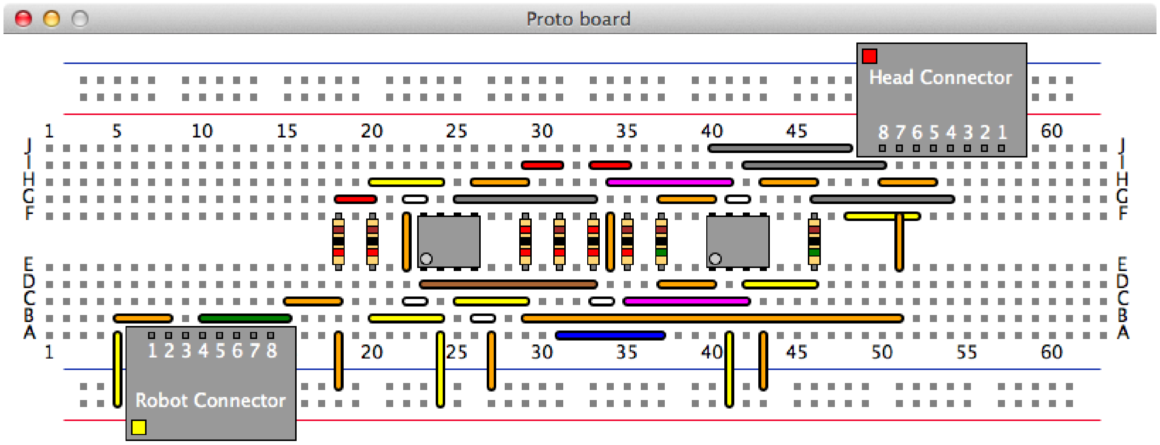


Figure 4-14: Exemplar for the per-node (decreasing) wiring method, using distance placement and $A*$ Search. We used distance placement instead of blocking placement to generate this exemplar because the combination of blocking placement with this wiring method consistently failed on the schematic shown in Figure 4-3.

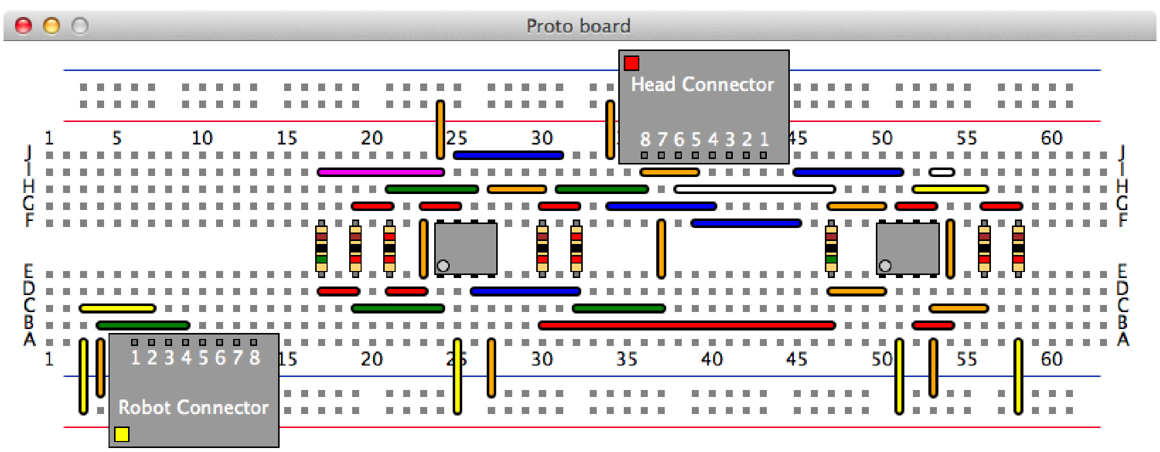


Figure 4-15: Exemplar for the per-pair (increasing) wiring method, using blocking placement and $A*$ Search.

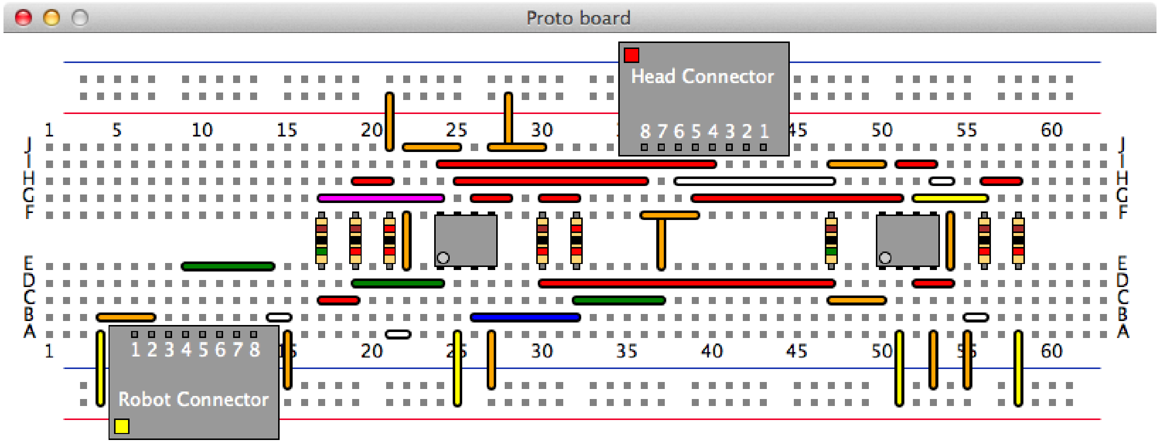


Figure 4-16: Exemplar for the per-pair (decreasing) wiring method, using blocking placement and $A*$ Search.

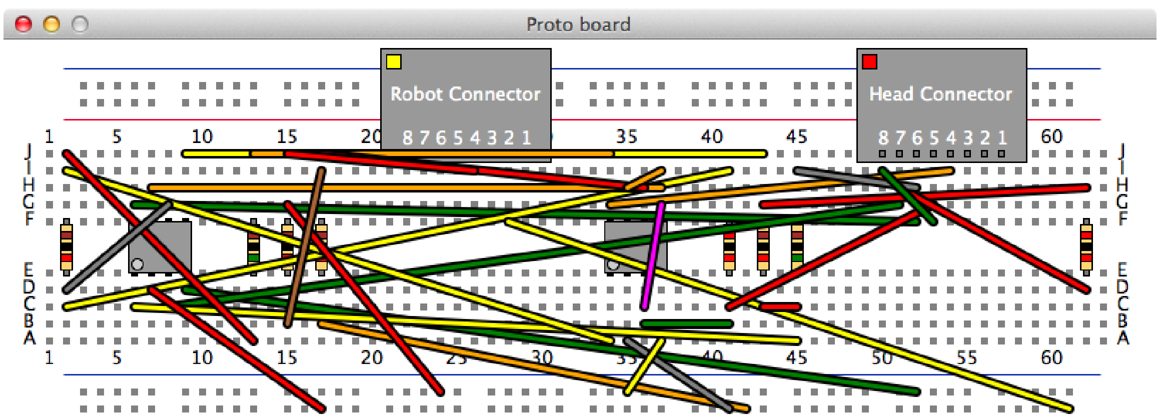


Figure 4-17: Exemplar for the straight wiring method, using blocking placement.

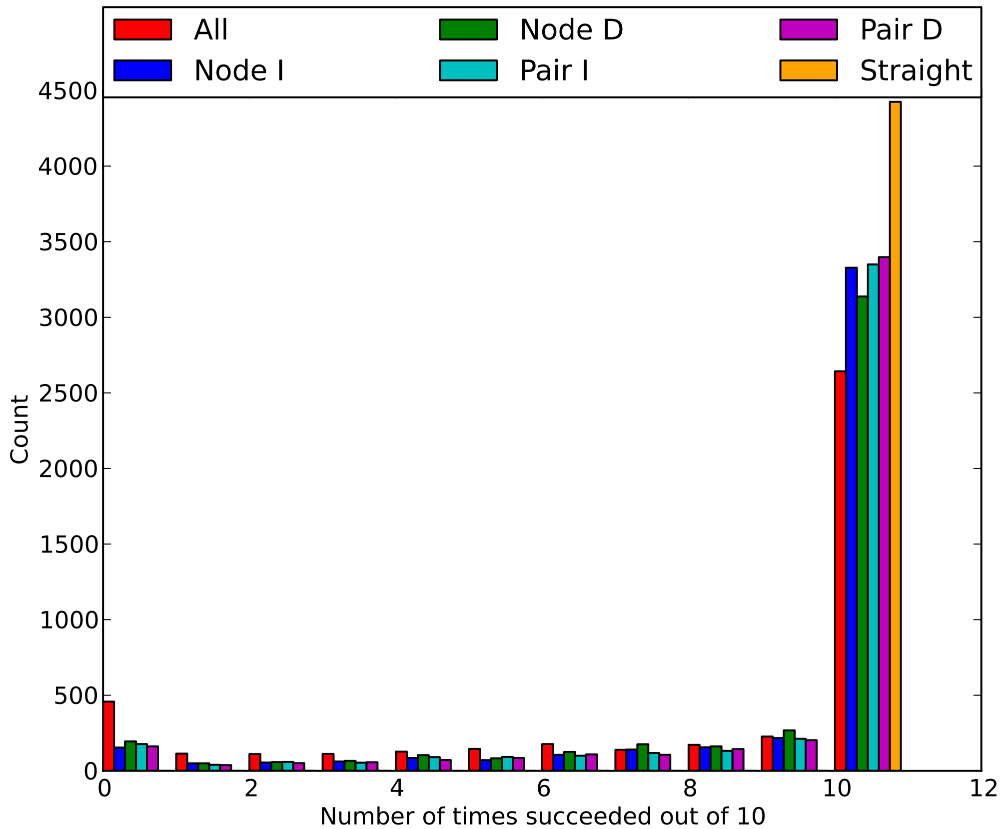
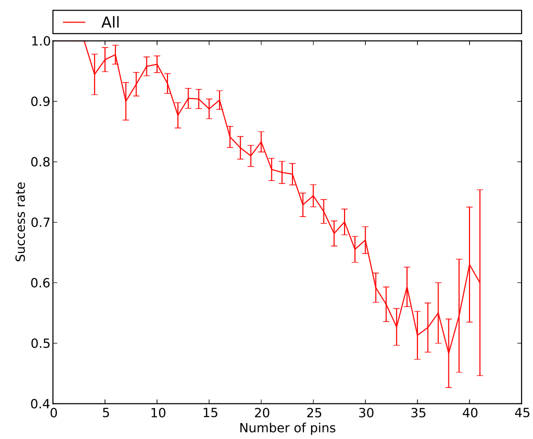


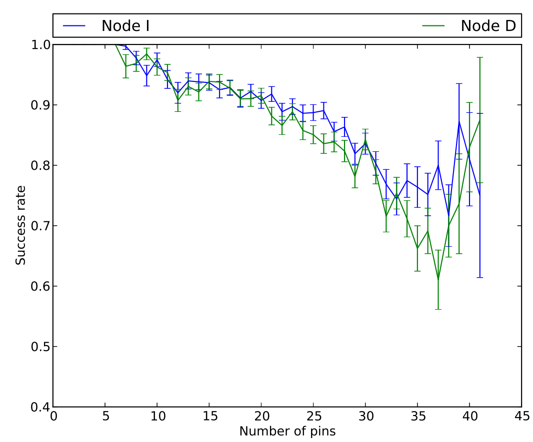
Figure 4-18: Wiring method success rate comparison.

	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

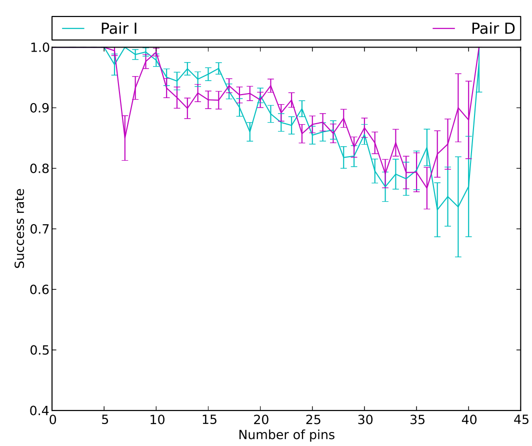
Table 4.2: Wiring method success rate comparison.



(a) All pairs



(b) Per-node



(c) Per-pair

Figure 4-19: Wiring method success rate trend comparison.

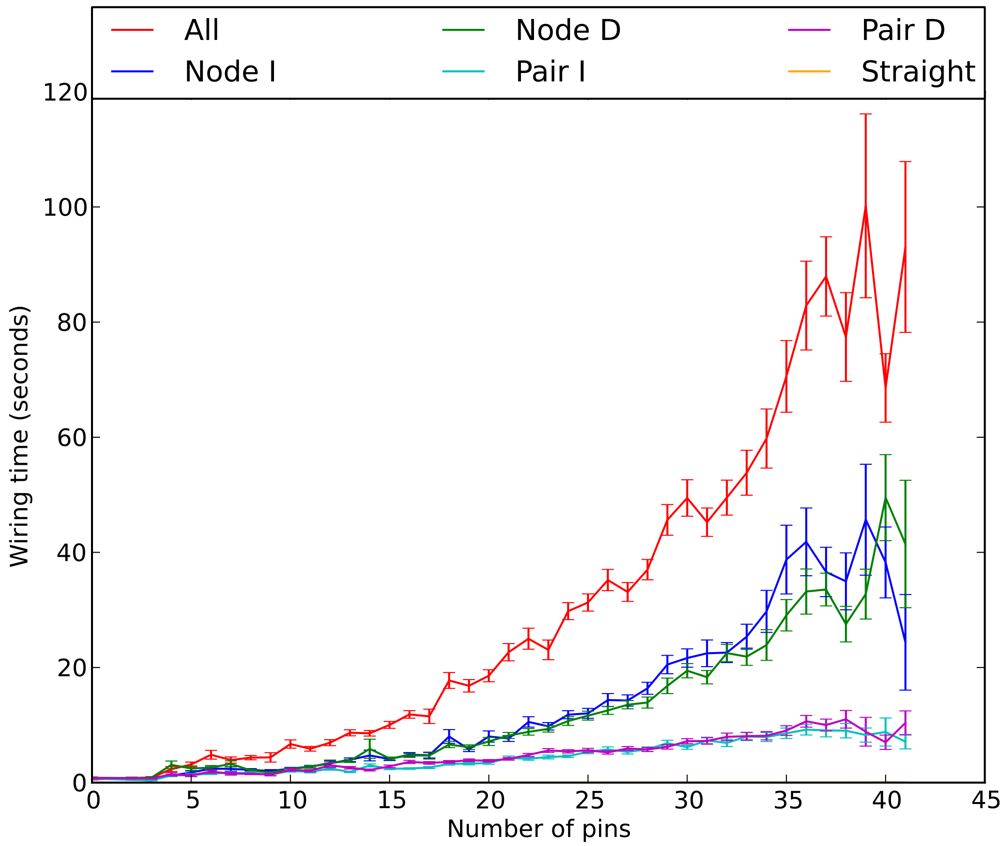


Figure 4-20: Wiring method wiring time trend comparison for successful runs. Note that the line for Straight wiring is not visible because it is so close to 0 for all values of circuit complexity.

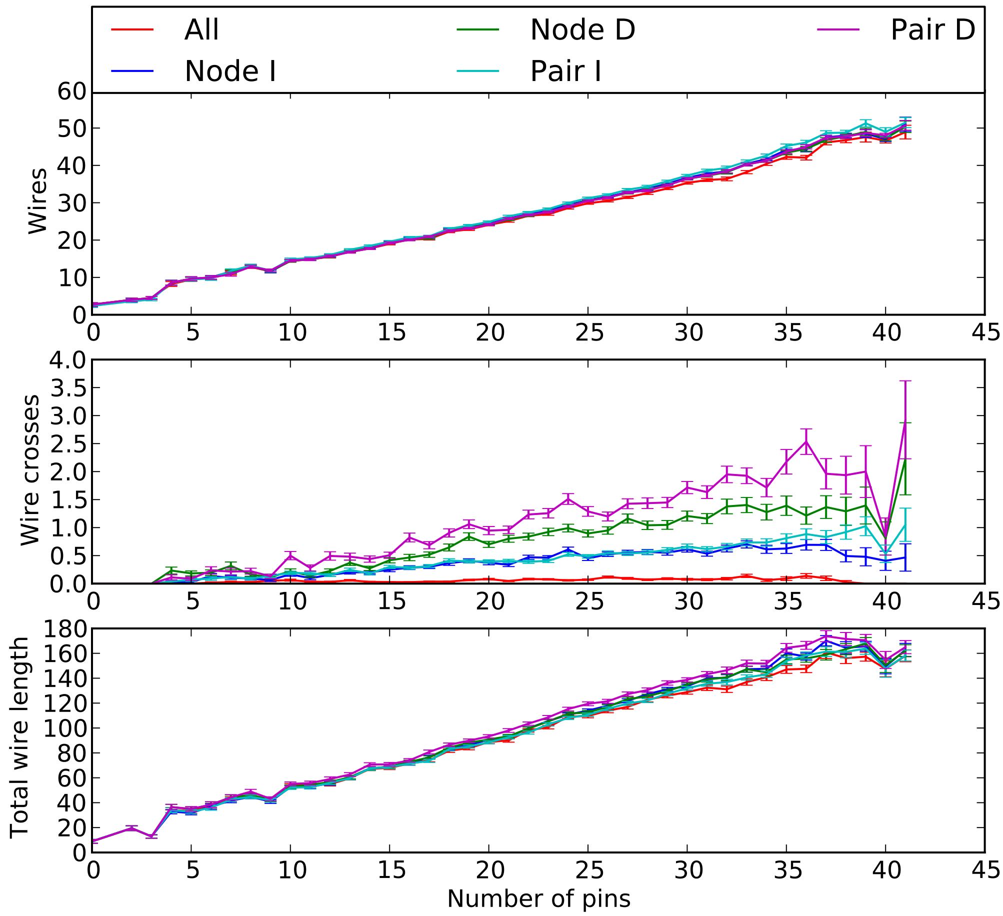


Figure 4-21: Wiring method layout quality trend comparison.

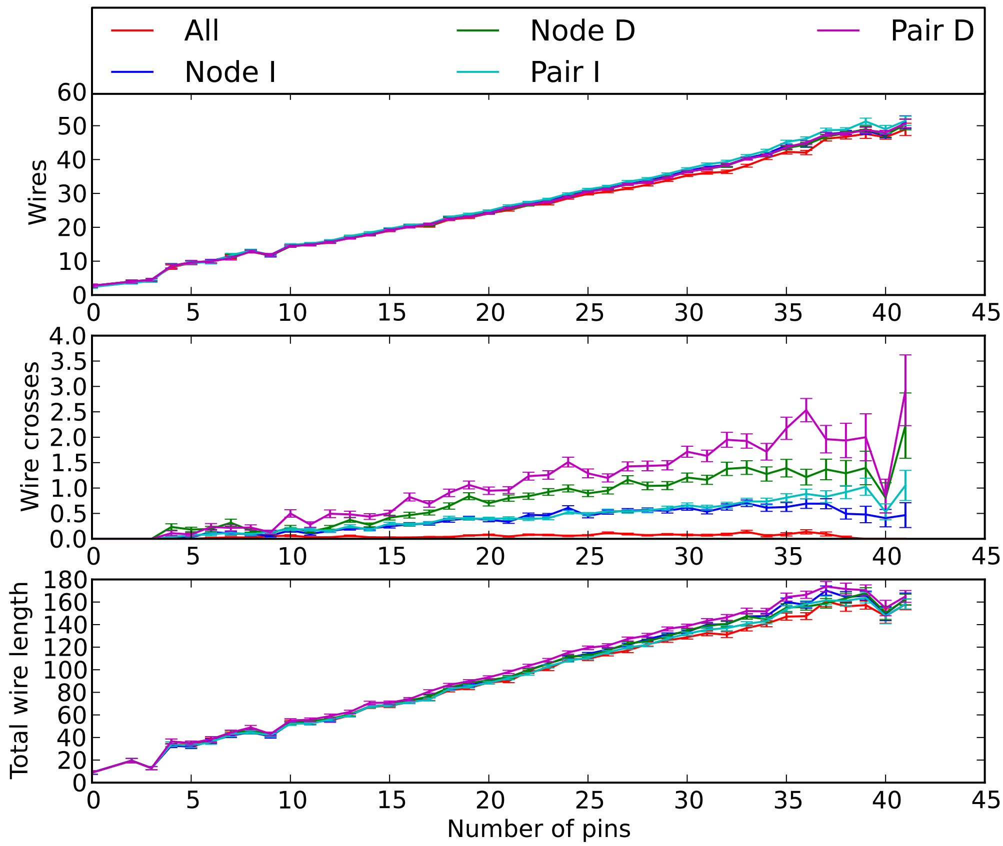


Figure 4-22: Wiring method layout quality trend comparison, not including the Straight wiring method.

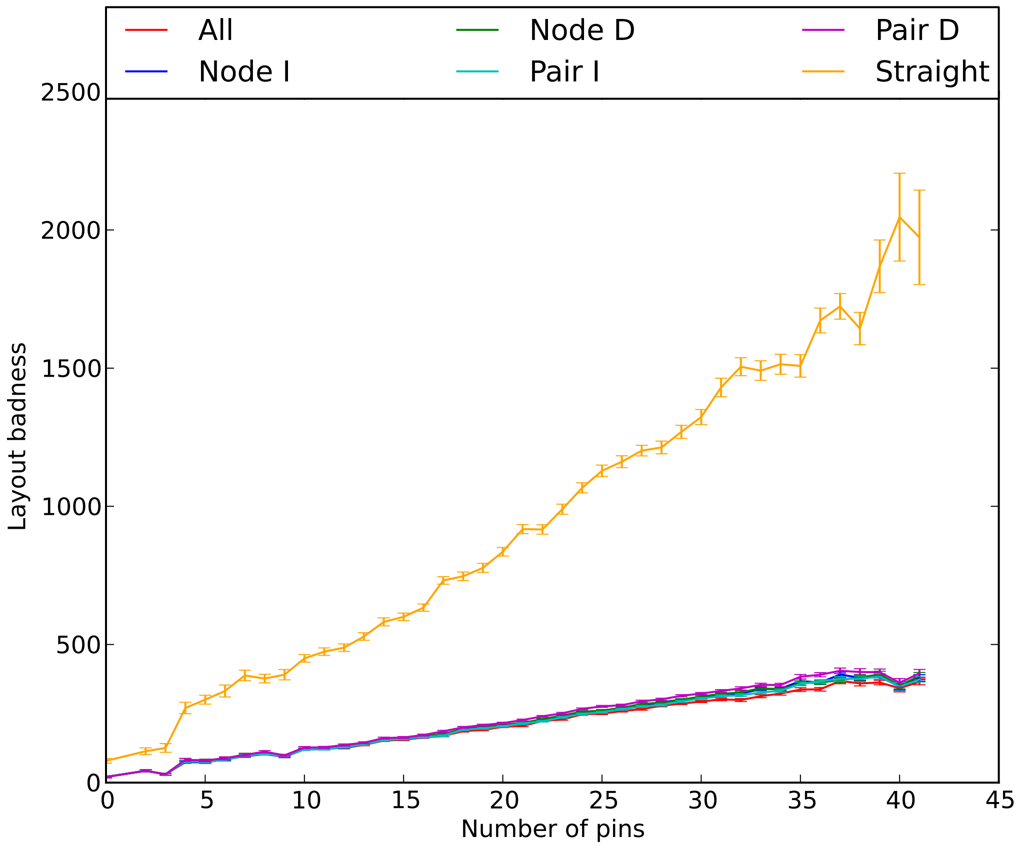


Figure 4-23: Wiring method layout badness trend comparison.

4.4 Comparing Search Methods

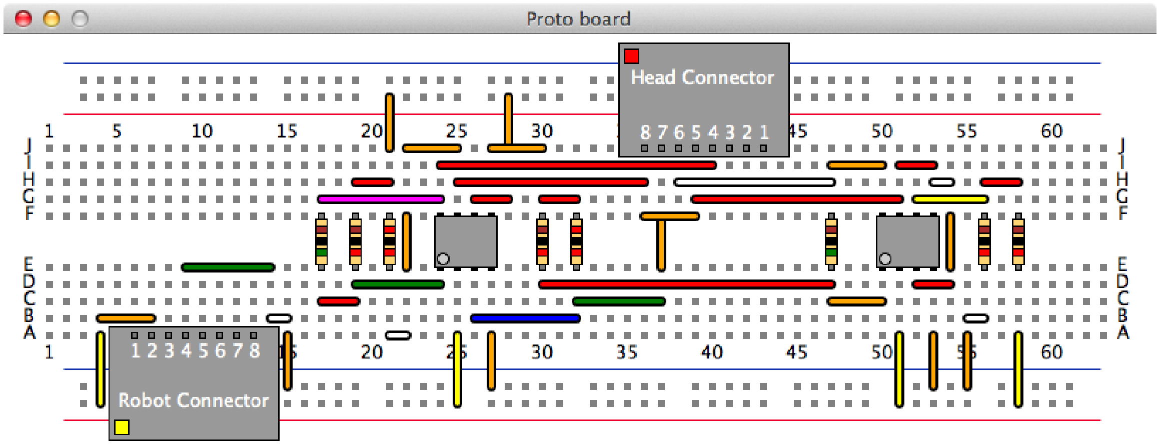


Figure 4-24: Exemplar for $A*$ Search, using blocking placement and per-pair (decreasing) wiring.

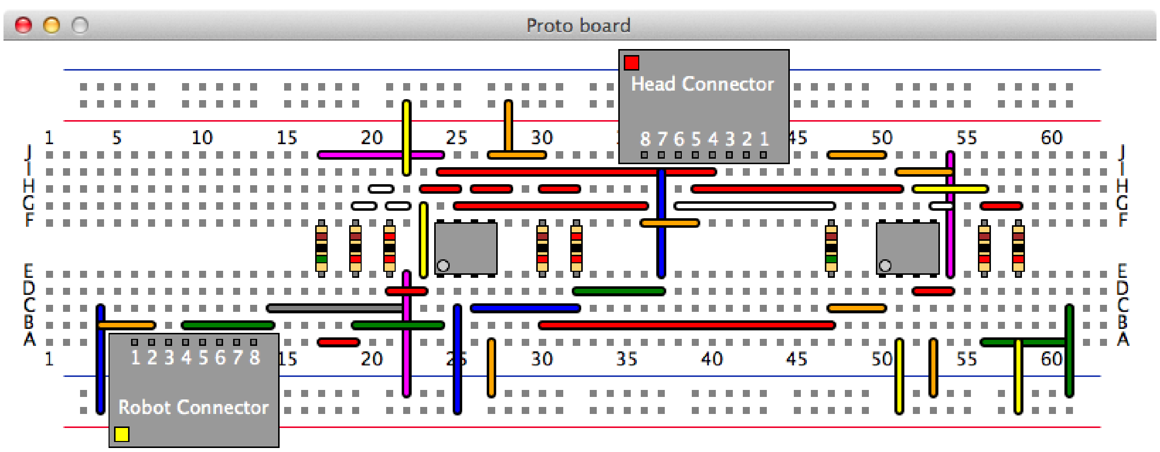


Figure 4-25: Exemplar for Best First Search, using blocking placement and per-pair (decreasing) wiring.

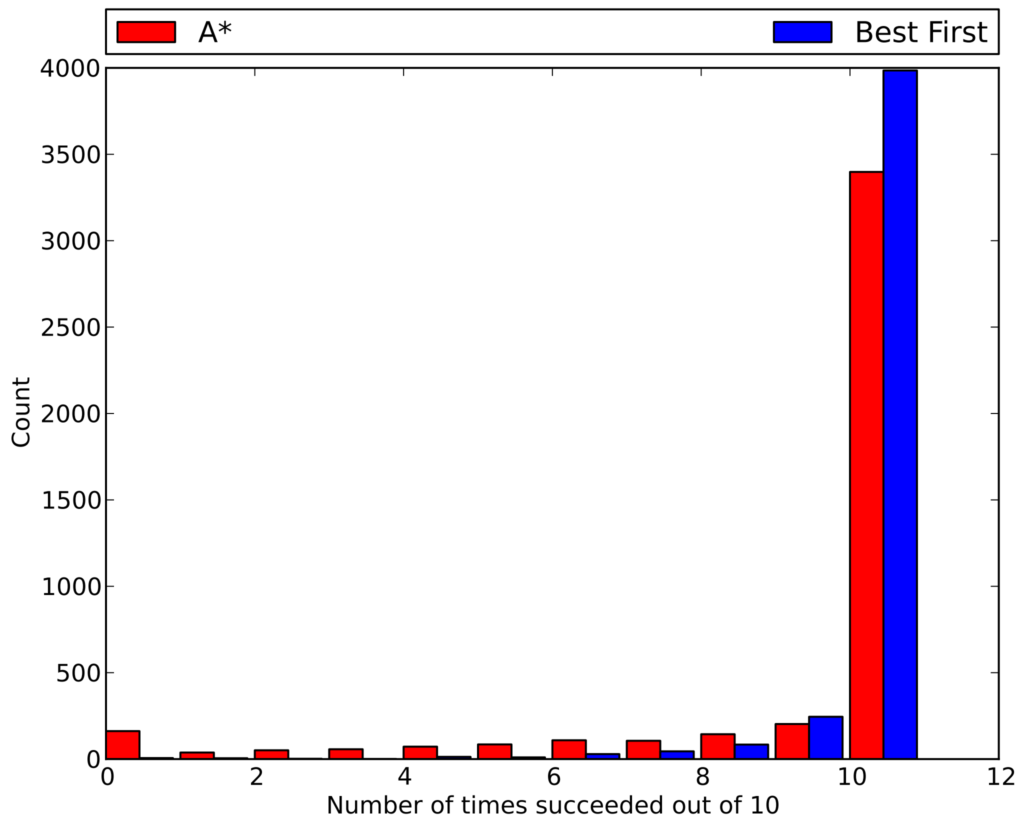


Figure 4-26: Search method success rate comparison.

	Number of times succeeded out of 10										
	0	1	2	3	4	5	6	7	8	9	10
$A*$	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
Best First	6 0.00	5 0.00	2 0.00	1 0.00	13 0.00	10 0.00	29 0.01	45 0.01	84 0.02	245 0.06	3985 0.90

Table 4.3: Search method success rate comparison.

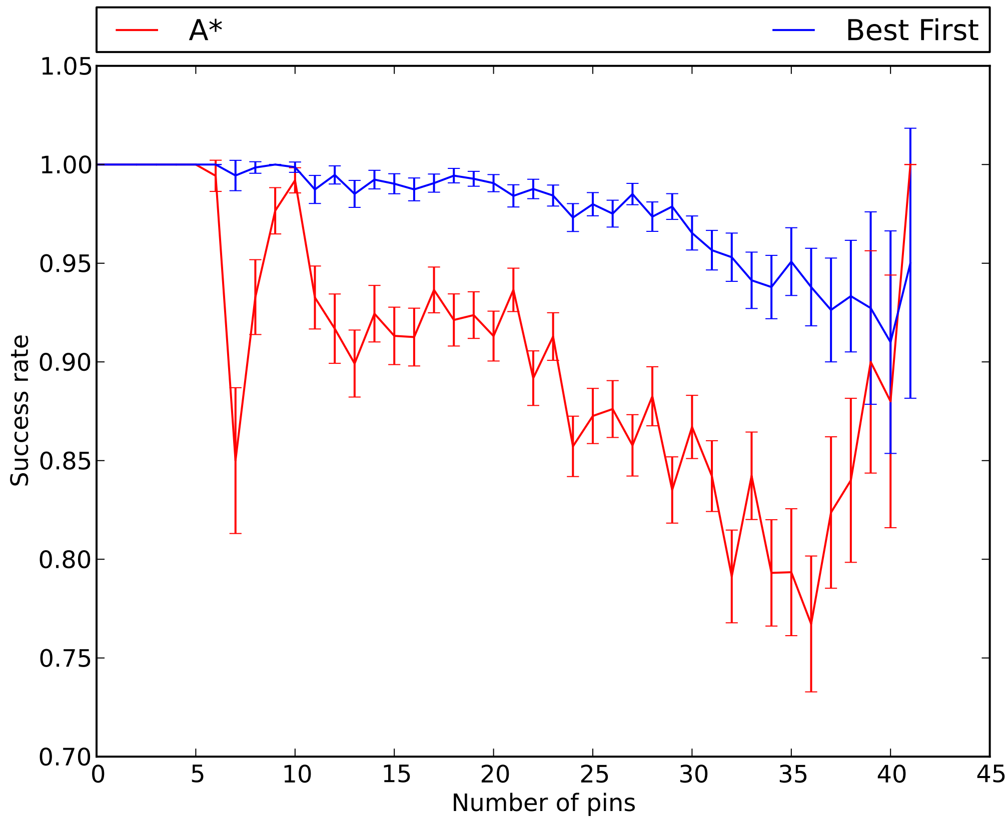


Figure 4-27: Search method success rate trend comparison.

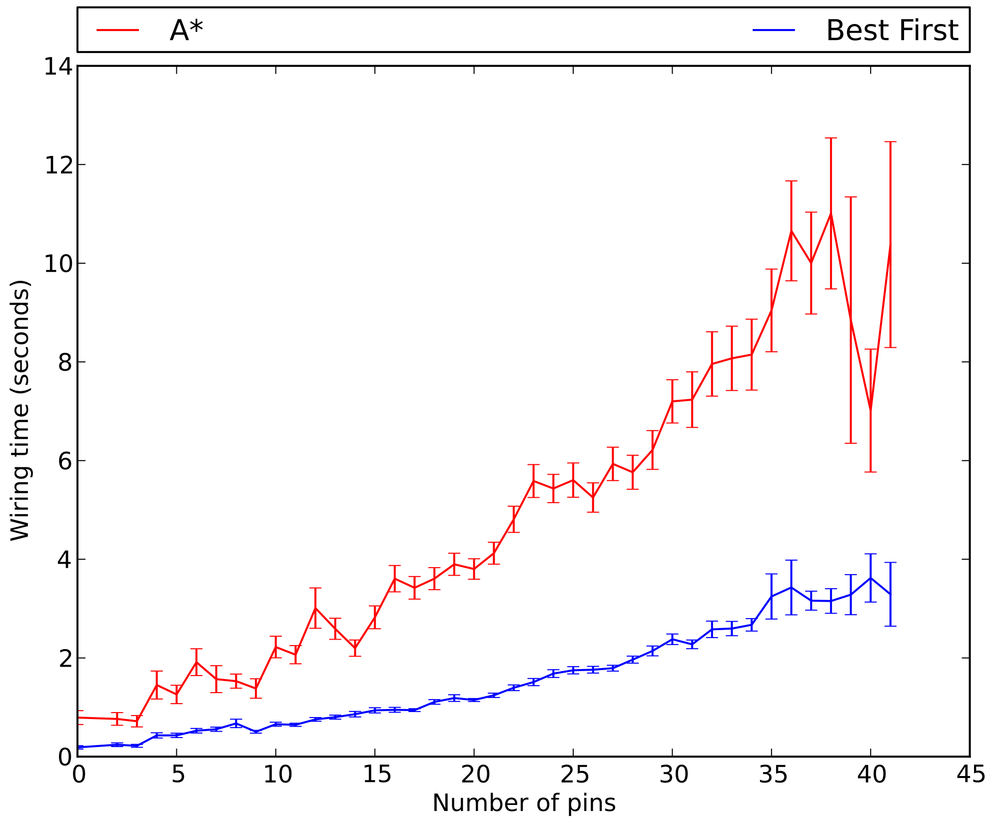


Figure 4-28: Search method wiring time trend comparison for successful runs.

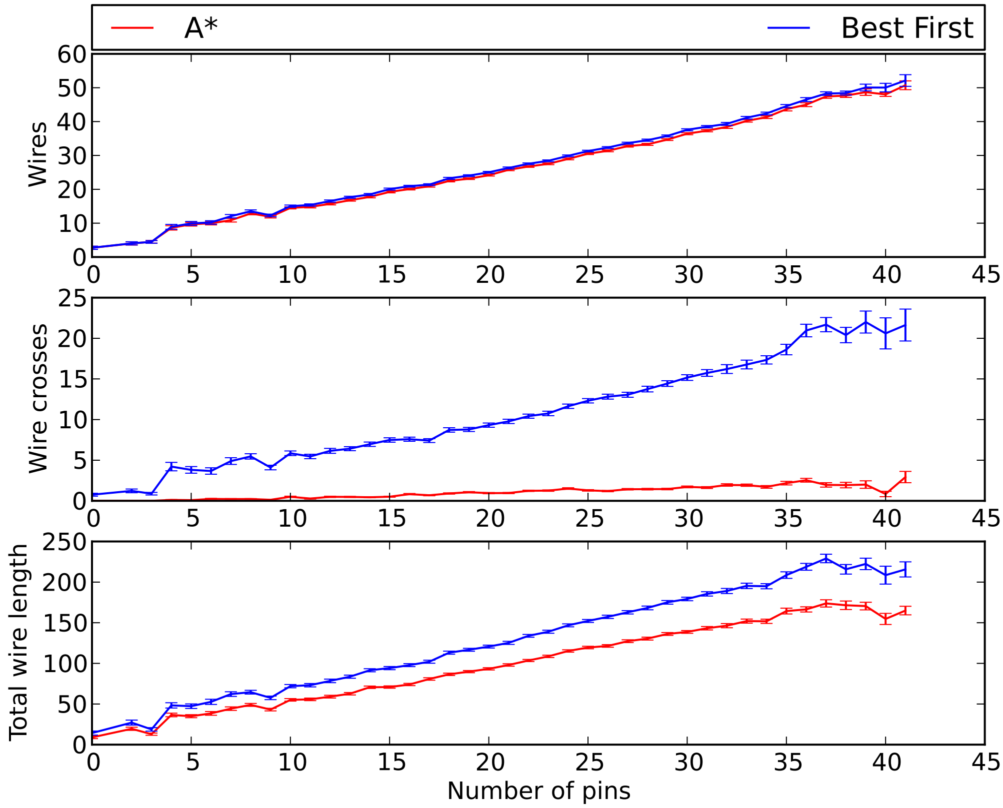


Figure 4-29: Search method layout quality trend comparison.

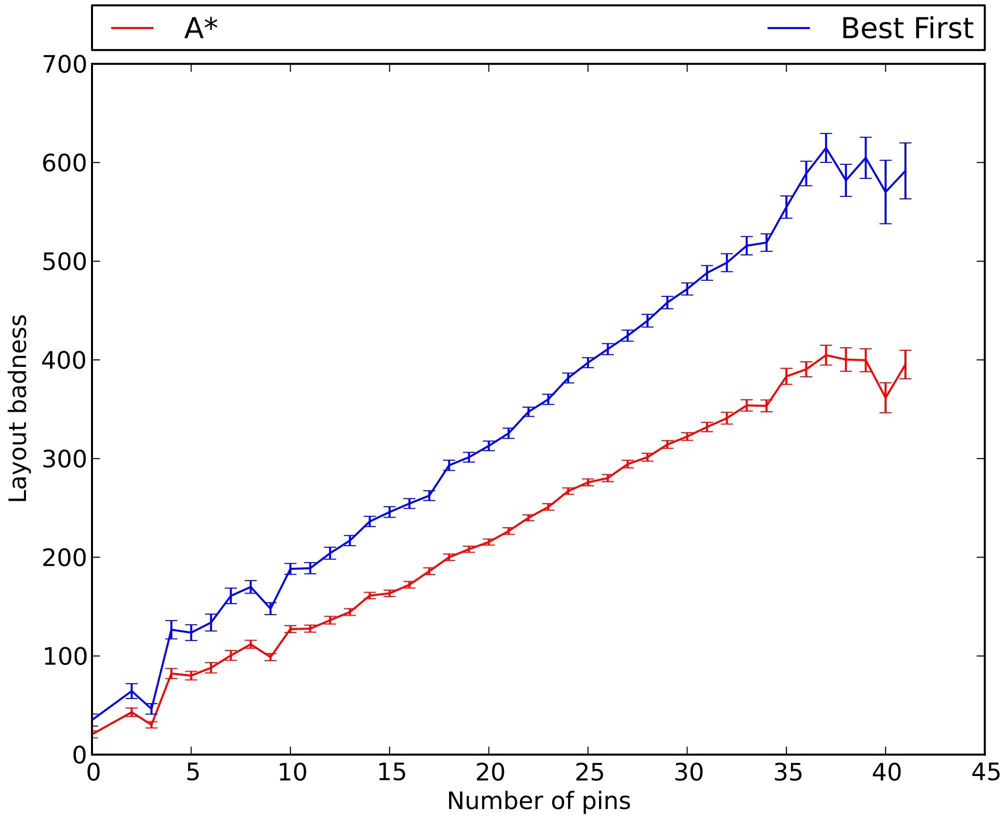


Figure 4-30: Search method layout badness trend comparison.

4.5 Combined Algorithm

Here we provide data for the combined algorithm presented in Section 3.2.3. To generate this data, we used a different dataset of 4425 schematics. As desired, the algorithm has a 100% success rate. Figure 4-32 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 connect a few pairs of locations by putting down a straight wire bridging the locations. Figure 4-34 gives the average total time taken by the algorithm as a function of circuit complexity. Finally, Figures 4-35 and 4-36 give statistics on the quality of the layouts produced by the combined algorithm as a function of circuit complexity.

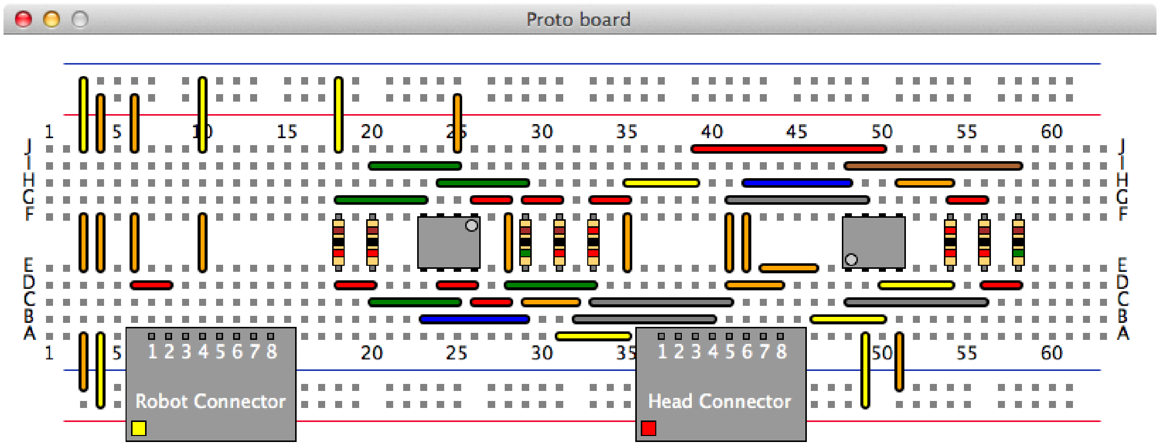


Figure 4-31: Combined algorithm exemplar. Notice that the top rail rows (the first and second row) are forced to be used as power and ground nodes.

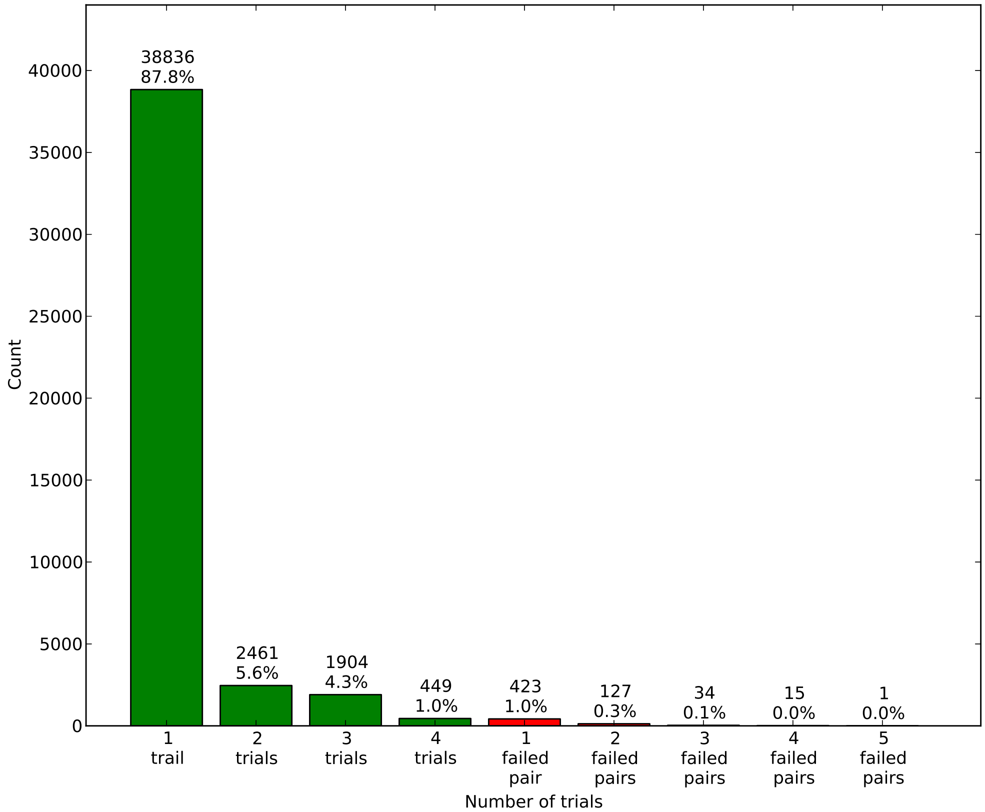


Figure 4-32: Combined algorithm success summary.

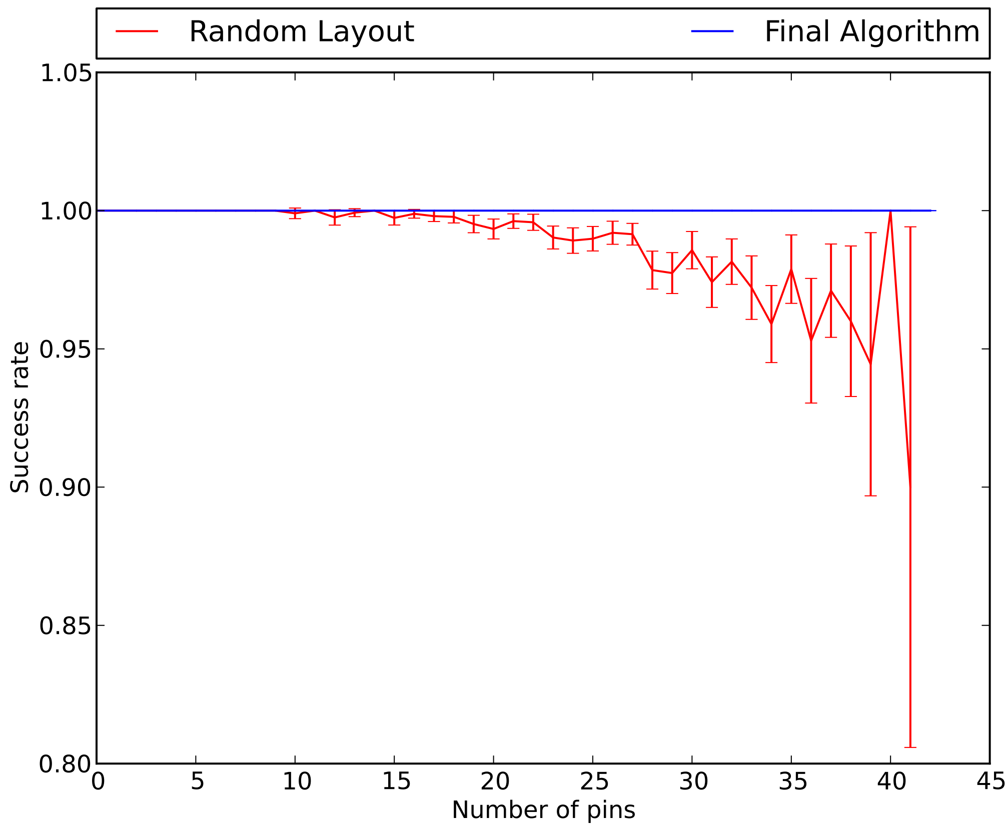


Figure 4-33: Combined algorithm success rate compared to random layout success rate as a function of circuit complexity.

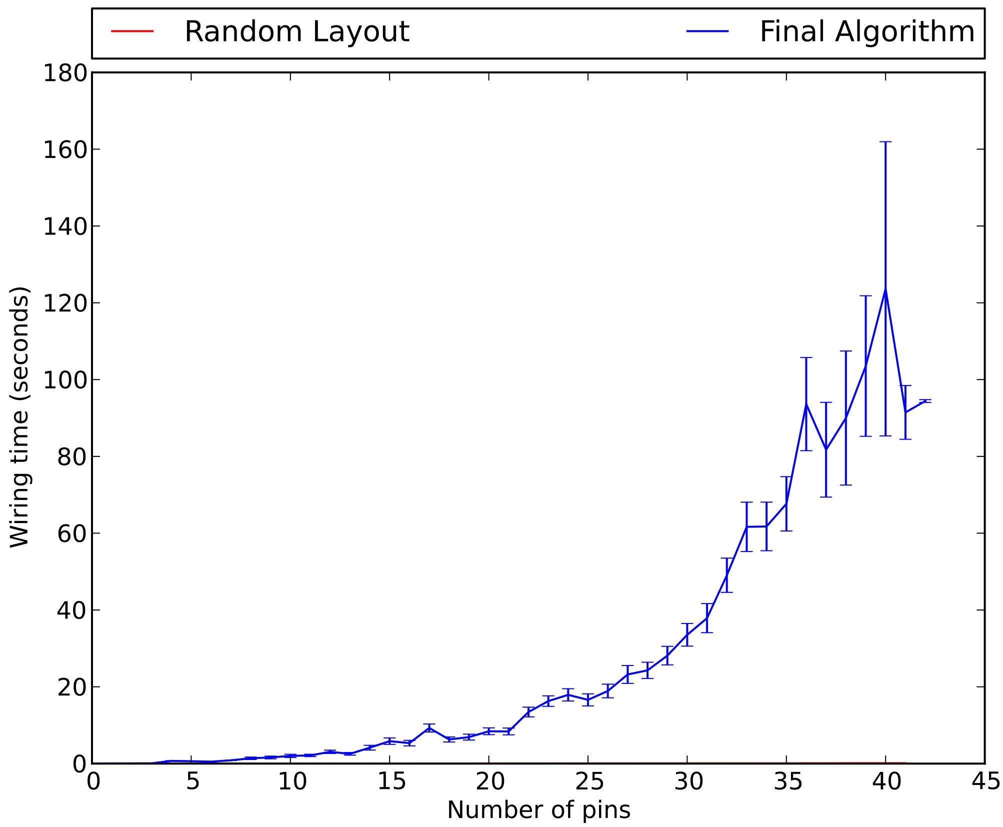


Figure 4-34: Combined algorithm total CPU time trend as a function of circuit complexity. Note that the line for Random Layout is not visible because it is so close to 0 for all values of circuit complexity.

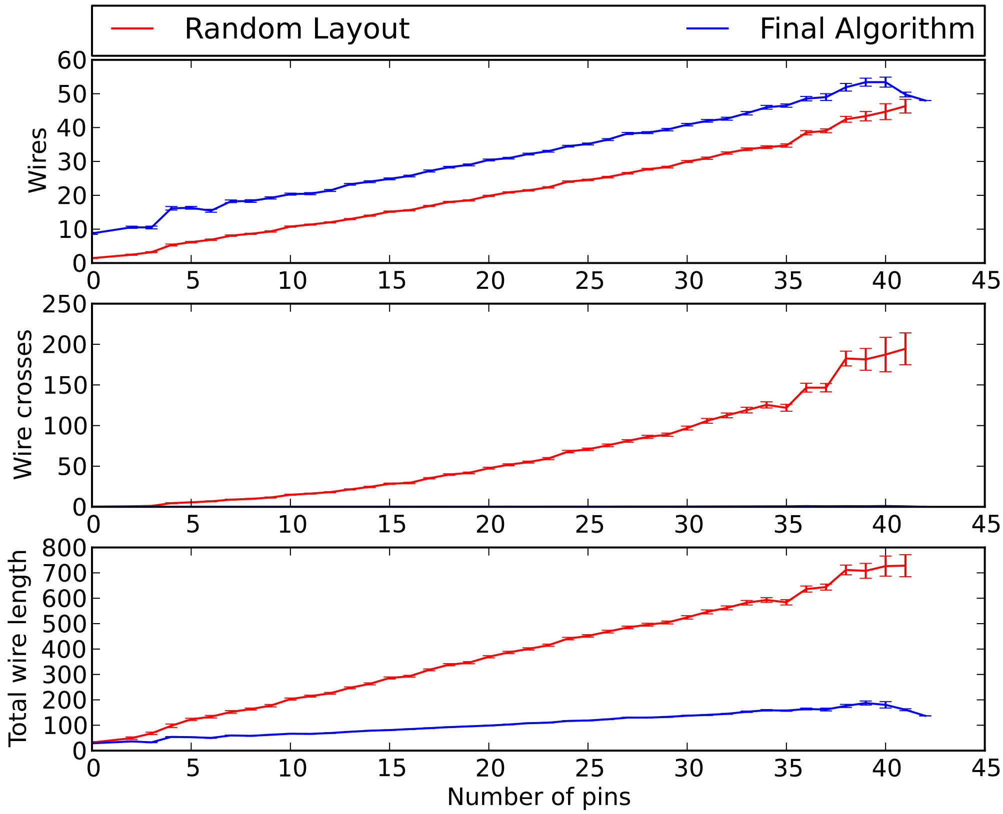


Figure 4-35: Combined algorithm layout quality trend. Note that the line for number of wire crosses for the Final Algorithm is not visible because it is so much closer to 0 than the number of wire crosses for Random Layout for all values of circuit complexity.

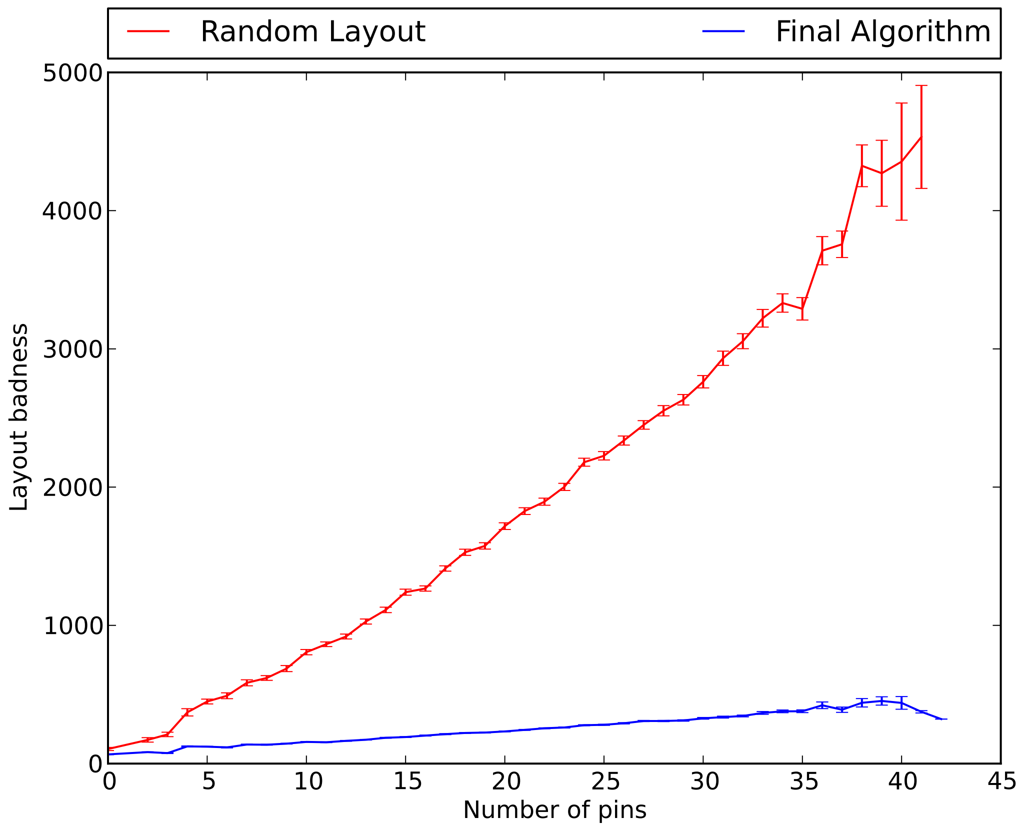


Figure 4-36: Combined algorithm layout badness trend.

Chapter 5

Discussion

In this chapter we provide justifications for the choices made in solving the protoboard layout problem, as well as 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 down wires on an empty protoboard (even in ways that may not make sense from a circuit theoretic 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 givens an expression for $T(n)$[1].

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

In this problem, we have that $n = 830$, the number of available locations on an empty protoboard. Evaluating T at $n = 830$ yields approximately 2.8×10^{1043}. The

[1]The sequence of numbers described by $T(n)$ is sometimes referred to as the telephone numbers or the involution numbers.

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 size, we place resistors only in the middle strip of the protoboard, as shown in Figure 3-2. This choice 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, this restriction is bad as it imposes a restriction on the size of the schematics (in terms of the number of components) for which a layout can be generated using the algorithm. 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 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. All together, there are 3 different ways to package together 2 op-amps[2]. Table 5.1 gives the number of different packagings possible for various n.

$$\text{Number of ways to package } n \text{ op-amps} = \sum_{k=0}^{\lfloor \frac{n}{2} \rfloor} \frac{n!}{k!(n-2k)!} \tag{5.2}$$

[2] When using only one of the op-amps in an op-amp package, we assume that we use the one on the left as drawn in Figure 2-2(c).

n	Number of ways to package n op-amps
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.

Our placement approach explores all possible ways of packaging 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 is not too computationally intensive. On circuits with more than 6 op-amps, this approach quickly becomes intractable, as the number of alternatives to consider would be far too large, and we would have to consider different strategies.

5.3 Explaining the Results

Chapter 4 presented quantitative data to compare alternative strategies for solving the protoboard layout problem. Here, we analyze those data and give reasonings for why we obtained those results.

5.3.1 Comparing Placement Methods

Success Rate

The blocking placement method (89.0% overall success rate across the 44250 runs) is slightly more successful than the distance placement method (87.6% overall success rate). While these alternative methods are not markedly different in terms of success rate, we note that both methods are more successful than the random placement alternative (43.5% overall success rate). As a function of circuit complexity, Figure 4-8 suggests that the two alternatives have almost identical success rates. As we would expect, success rate generally decreases for both of the placement methods as circuit complexity increases. Once again, we observe that the distance and blocking based placement methods are much more successful than the random placement method, especially as circuit complexity increases. The rise in success rate for high circuit complexity is a result of the fact that there are very few circuits of the highest complexity, on which the algorithm happened to be consistently successful.

Wiring Time

We observe from Figure 4-9 that, once again, the two methods are very similar, with random placement being markedly worse than both. We see that the distance method generally results in a layout for which the wiring takes less time than does the blocking method, but the difference between the two is almost negligible. As we would expect, we see that as the complexity of the circuits increases, the amount of time spent in the wiring step also increases. As circuit complexity increases, the standard error of wiring time when using random placement also increases. This is a result of the fact that random placement frequently fails on the most complex circuits, and so the sample size we have for successful runs is very small.

Layout Quality

Figure 4-10 presents graphs that compare numbers of wires, numbers of wire crosses, and total wire lengths. Figure 4-11 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. We see that the blocking method consistently results in more wire crosses. When we consider total wire length, the blocking method exceeds the distance method consistently, with the difference getting higher as circuit complexity increases. This is expected because the blocking method does not directly aim to put pieces that need to be connected close together, whereas the distance method directly tries to minimize the total wire length that may be needed. In terms of the layout badness metric, we observe once again that the distance method is slightly better than the blocking method for almost all circuit complexity values. Finally, we observe that the random placement method, per our badness metric, produces much worse layouts than the other two methods when it does succeed.

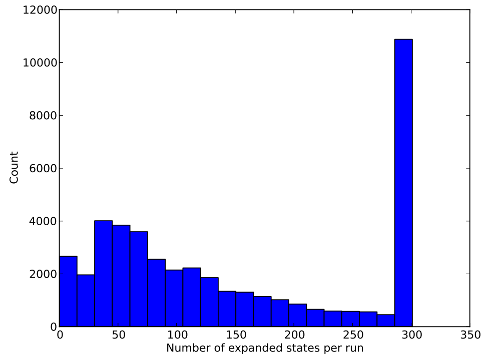
Conclusion

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

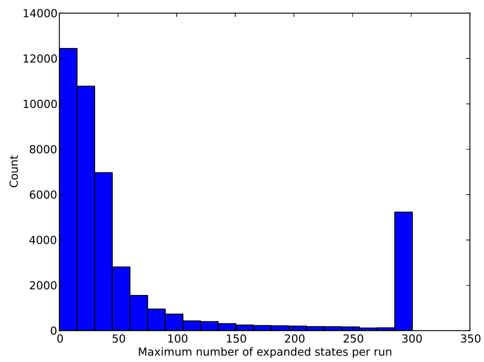
5.3.2 Comparing Wiring Methods

Success Rate

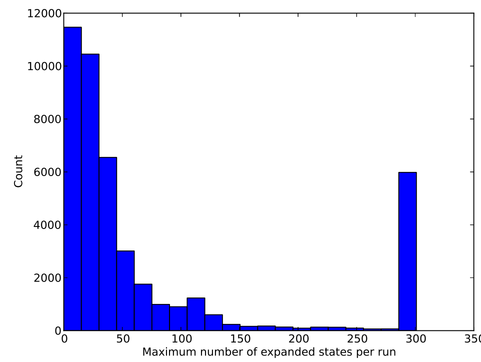
Figures 4-18 and 4-19, and Table 4.3 show that the all pairs wiring method has a much smaller success rate than all of the other alternatives, especially as circuit complexity increases. The reason why the all pairs method has a smaller success rate is in large part due to the limit on expanded vertices in $A*$. Figure 5-1 depicts histograms of the maximum number of states expanded in a search for each of the $4425 \times 10 = 44250$ test runs. In the case of all pairs wiring, the histogram is simply of the number of states expanded in the search for each of the test runs. In the other methods, multiple searches may be carried out per run, and we chose to look at the most expensive search in our analysis to understand the worst-case performance of the alternative methods. The figures depict that the limit on expanded vertices affects the all pairs wiring method much more than it does the others. This result is expected because the search task in all pairs wiring is more difficult than in the other alternatives. Notice that as the search tasks get less difficult from all pairs to per-node to per-pair wiring, the effect of the limit decreases. Hence, the other four alternatives have very comparable success rates. Finally, we note that straight wiring, as expected, has a 100% success rate.



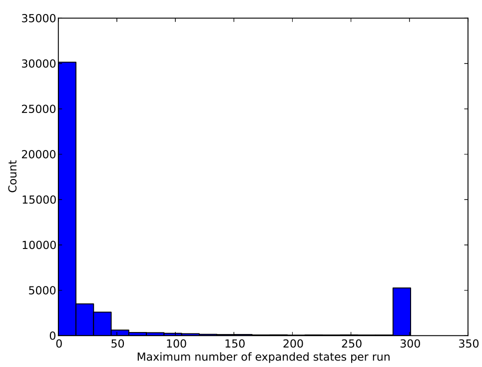
(a) All pairs



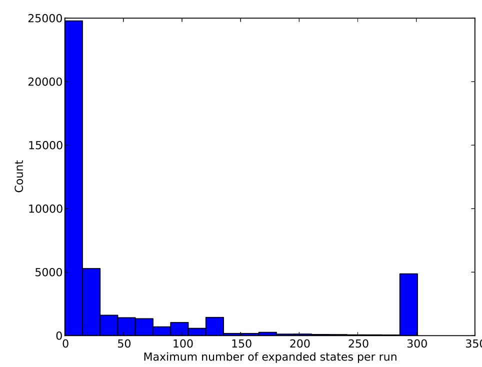
(b) Per node (increasing)



(c) Per node (decreasing)



(d) Per pair (increasing)



(e) Per pair (decreasing)

Figure 5-1: Histograms of maximum number of vertices expanded in $A*$ per test run. In the case of all pairs wiring, this value is simply the number of vertices expanded in the search.

Wiring Time

Figure 4-20 compares wiring time across the six methods. Once again, we observe that the all pairs methods takes more time than the other methods. We also observe from Figure 4-20 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, which generally requires searching through more alternatives than connecting each of the pairs of locations individually. Finally, we note that straight wiring, as expected, takes less time to complete than all of the other alternatives.

Layout Quality

Figures 4-22 and 4-23 compare the quality of the layouts produced by the six 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 among the alternatives that utilize search to accomplish the wiring. We note that straight wiring uses uses fewer wires than all of the other alternatives as it uses exactly one wire per pair of locations. There are noticeable 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 expected since the algorithm runs one search to connect all pairs of locations. Conversely, the per-pair (decreasing) and per-node (decreasing) methods result in the largest number of wire crosses. Note that the per-node (decreasing) method produces more wire crosses on average than the per-pair (increasing) method. We observe that the order in which we consider pairs of locations has an effect on how good the generated layouts will be. In essence, connecting the harder pairs of locations first generally produces more wire crosses. Finally, as expected, we observe that straight wiring produces comparatively poor layouts.

Conclusion

While the all pairs method is the least successful method and generally takes the longest among the six, it tends to produce the best layouts when it does succeed. On the other hand, the alternatives that break the problem down into smaller pieces succeed more often and finish more quickly, though they tend to produce worse results. Furthermore, the more finely we break down the problem, the faster the overall algorithm runs. Lastly, ordering subproblems from hardest to easiest has the effect of making the overall wiring step faster, but produces worse results than the reverse order and does not result in a markedly better success rate.

5.3.3 Comparing Search Methods

Success Rate

Figures 4-26 and 4-27 and Table 4.4 present data comparing the success rates of the two search algorithms. We 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, when using Best First Search, the algorithm looks for layouts that satisfy the connection requirements, ignoring the badness of the layouts it considers, whereas $A*$ searches for an "optimal" layout. Hence, Best First Search is less susceptible to the restriction on number of vertices to expand than is $A*$.

Wiring Time

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 4-28 supports this expectation.

Layout Quality

Figures 4-29 and 4-30 show that the layouts generated by the algorithm when using Best First Search are worse than the layouts generated when using $A*$. 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.

Conclusion

Our choice of a search algorithm forces us to consider a trade-off between speed, success rate, and quality. Using Best First Search, most runs will be successful and terminate quickly, but will produce very poor results. Using $A*$, fewer runs will be successful, and the successful runs will take longer to terminate, but the resulting layouts will generally be better.

5.3.4 Combined Algorithm

In this section we discuss the structure of Algorithm 2, and we also discuss the data we obtained for the combined algorithm. Recall that the combined algorithm makes four attempts at generating a complete layout. The algorithm tries both placement methods, using the distance method first and the blocking method second. It tries the distance method first because the distance method tends to generate layouts with fewer crossing wires and smaller total wire length. The algorithm uses per-pair wiring, and consider both orders of doing the wiring, increasing order first and decreasing order second. It uses per-pair wiring because per-pair wiring takes considerably less time than both per-node and all pairs wiring, and neither of the other wiring methods has a better success rate. The algorithm tries increasing order first because increasing order tends to generate layouts with fewer crossing wires than the reverse order.

Let us now consider the data we obtained for the combined algorithm. Firstly, we see 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. This success rate is a result of the last part of Algorithm 2 that uses single wires to connect pairs of locations that need to be connected. Figure 4-32 shows that, on the test dataset of 4425 schematics, with 10 runs carried out on each schematic, the algorithm succeeded in generating layouts for 98.6% of the circuits without having to put down forced wires (wires added at the last step of the algorithm to connect any disconnected location pairs). The algorithm was required to put down more than 2 forced wires on only 0.1% of the circuits. From Figure 4-34, 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 (which is most notable as the number of op-amps increases) as well as a more difficult wiring task. Importantly, 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 standpoint because 3 minutes is not long for a student to have to wait for a layout to be generated. Finally, Figures 4-35 and 4-36 present trends of quality as a function of circuit complexity.

5.4 Further Work

5.4.1 Treating Resistors as Wires

Our current solution treats resistors as circuit components and so they are placed before the wiring search is executed. However, resistors have the special property among the components that they can be placed as if they were wires of a fixed set of lengths. This suggests we can think of resistors as wires, and thus can handle them in the wiring step of the algorithm instead of the placement step. As a result, the search that we do in the wiring step would need to be more elaborate. Not only would we need to keep track of pairs of locations on the protoboard that need to be connected, but we would also need to know whether to put a resistor between the two locations. In the latter case, we 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, its implementation is not trivial. First, if there is any node in the 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 possible solution to this problem is to reserve an empty column on the protoboard for the node between the resistors, which we can then 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 make the search we carry out more complicated. Second, we would need a new kind of heuristic for the search that takes resistors into account. One possible solution is to highly penalize an absent resistor when computing the heuristic. Even with an amended heuristic, however, there are cases where the search may not find an answer (for instance a pair of adjacent locations that must be connected by a resistor).

5.4.2 Building Layouts Similar to Previously Generated Layouts

One problem with the current tool 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 the tool come in to play. As an alternative solution to this problem, it may be useful to have the tool remember the last placement it used and try to produce a new placement as similar to the last one as possible.

5.5 Remarks

This paper introduced a circuit schematic entry tool for 6.01 capable of automatically generating protoboard layouts. The final algorithm presented in this paper is able to generate layouts with a 100% success rate on our test dataset, and produces markedly better layouts than randomly generated ones based on our metric of layout badness. These results are encouraging as they suggest that the tool may be a good addition to 6.01 as a compliment, or perhaps replacement, for CMax. If, as we intended, the tool's performance on the test schematics is any indication of its performance on schematics that 6.01 students may design, then the tool achieves the goal of avoiding the tedium of protoboard layout in 6.01 labs. Additionally, the schematic entry tool, which is also capable of circuit analysis at the same level as CMax, would make circuits schematics the main mode of communication in 6.01 labs (instead of protoboard layouts).

Appendix A

Schematic Entry 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, we describe the capabilities and features offered by the schematic entry GUI. Figure A-1 presents the GUI containing a sample schematic. The figure depicts the four important sections of the GUI: the palette, the board, 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 be included in 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 (the board is described in detail in Section A.2). 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 the robot and head connectors, respectively. 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 shows these last two circuit components as they would appear in the editor.

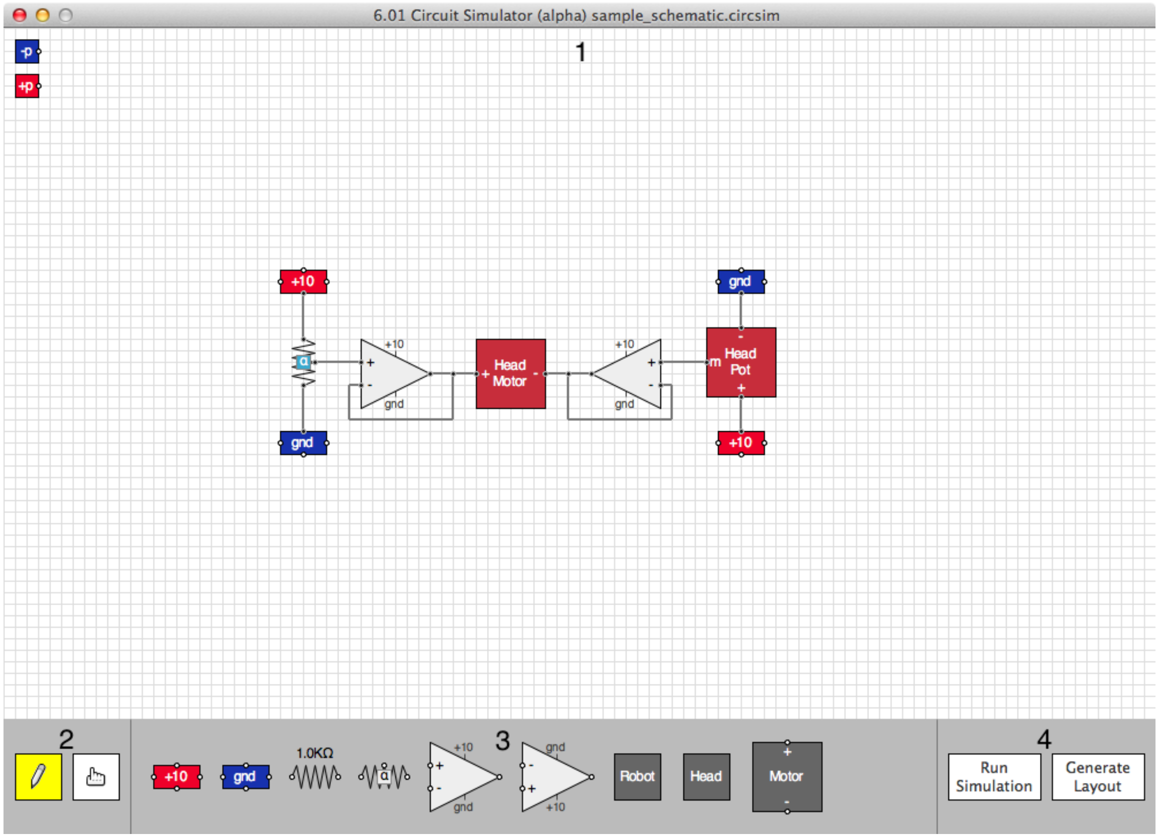


Figure A-1: Parts of the schematic entry GUI. Part 1 is the board on which the user constructs the schematic. Part 2 allows the user to toggle the cursor state. Part 3 is the palette from which the user can take circuit components. Part 4 presents the simulation and layout buttons.

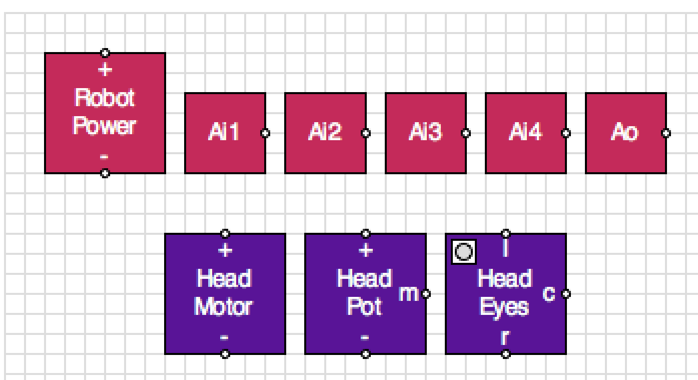


Figure A-2: The robot connector and head connector are presented as grouped components in the GUI.

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 edges 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 draw a wire to connect components by clicking on a connection point on a component and dragging. A wire may be drawn to another connection point, or a wire already on the board (which snaps the new wire onto the existing wire), or an empty location on the board (which creates 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 state 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, or dragging components, the GUI attempts to route the wires in a way that is aesthetically pleasing. That is, the wires are routed so as to avoid crossing wires and, more importantly, wires crossing components on the board. This routing 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, so circuits are simulated exactly as they would be 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 corresponding lamp's 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.

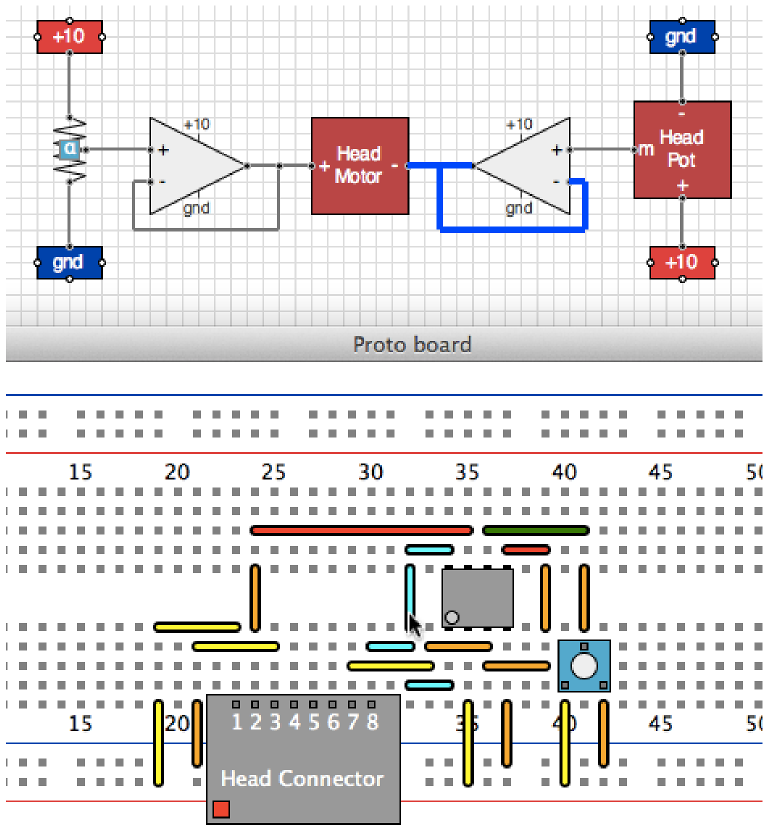


Figure A-3: Wire highlighting example.

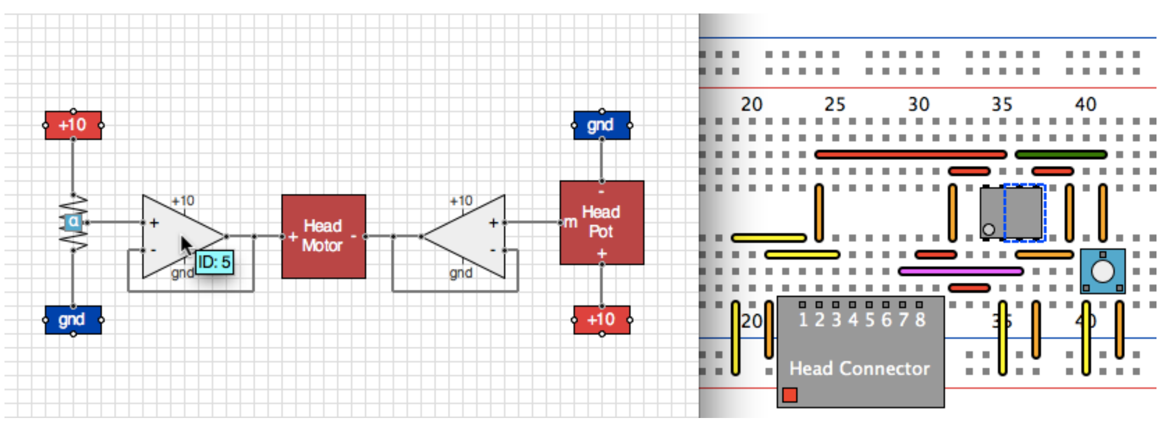


Figure A-4: Component highlighting example.

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
Ctrl + n	New file
Ctrl + o	Open file
Ctrl + s	Save file
Ctrl + q	Quit
Ctrl + z	Undo
Ctrl + y	Redo
Ctrl + w	Close simulation windows
g	Generate layout
s	Run simulation
Delete	Delete selected item(s)
r	Rotate selected item
d	Toggle cursor state
$\leftarrow$ | h	Move selected item(s) left
$\downarrow$ | j	Move selected item(s) down
$\uparrow$ | k	Move selected item(s) up
$\rightarrow$ | l	Move selected item(s) right

Table A.1: Shortcuts.

Bibliography

[1] Cadence. http://www.cadence.com.

[2] Eagle. http://www.cadsoftusa.com.

[3] Kaelbling, Leslie, Jacob White, Harold Abelson, Dennis Freeman, Tomás Lozano-Pérez, and Isaac Chuang. 6.01SC Introduction to Electrical Engineering and Computer Science I, Spring 2011. (MIT OpenCourseWare: Massachusetts Institute of Technology), http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-01sc-introduction-to-electrical-engineering-and-computer-science-i-spring-2011 (Accessed 7 Dec, 2013). License: Creative Commons BY-NC-SA.

[4] Introduction to EECS I. http://web.mit.edu/6.01.

[5] 6.01 Staff. CMax. http://web.mit.edu/6.01/www/cmax.