ABSTRACT

LAMMERT, ADAM CRAWFORD. Searching for Better Logic Circuits: Using Artificial Intelligence Techniques to Automate Digital Design. (Under the direction of Dr. Edward Willmore Davis Jr.)

Logic circuits are at the core of modern computing. The process of designing circuits which are efficient is thus of critical importance. Usually, logic circuits are designed by human beings who have a specific repertoire of conventional design techniques. These techniques limit the solutions that may be considered during the design process in both form and quality. The limits guide designers through the immense realm of possible circuits, thus making the problem more manageable. Simultaneously, the limits sometimes eliminate from consideration circuits which are optimal in terms of size, depth, etc. By exploring the full range of possible solutions, circuits could be discovered which are superior to the best known human designs. Automated design techniques borrowed from artificial intelligence have allowed exactly that. Specifically, the application of genetic algorithms has allowed the creation of circuits which are substantially superior to the best known human designs. This paper expands on such previous research with a three-fold approach. This approach is comprised of (1) two distinct optimizations for the application of genetic algorithms to design, (2) the formulation and implementation of a systematic search technique to the problem and (3) a comparison of the relative merits of the optimized genetic algorithm and the systematic search technique. It is contended that both genetic algorithms and systematic search can be preferable depending on the situation at hand.

SEARCHING FOR BETTER LOGIC CIRCUITS: USING ARTIFICIAL INTELLIGENCE TECHNIQUES TO AUTOMATE DIGITAL DESIGN

by
ADAM CRAWFORD LAMMERT

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

COMPUTER SCIENCE

Raleigh, NC

2006

APPROVED BY:

_____ _____

_____
Chair of Advisory Committee

BIOGRAPHY

Adam Crawford Lammert was born in Pittsburgh, Pennsylvania and raised in Western New York, near Buffalo. His interest in science developed early on, and has always been highly varied. As an undergraduate at Vassar College, his interest was captured by the interplay of mind and machine. As a result, he studied both Cognitive Science and Computer Science. His research at Vassar focused on robotic implementations of intelligence. After graduating, he moved south to attend North Carolina State University. There, he studied primarily Computer Science. The research at NCSU surrounded Artificial Intelligence, and its application to circuit design. That research culminated in this thesis. Currently, Adam is living in the San Francisco Bay Area and working as a laboratory assistant in the field of auditory perception.

ACKNOWLEDGEMENTS

I would like to acknowledge and offer my sincerest appreciation to Dr. Edward Davis for his intellectual and editorial support throughout this process, as well as his consistent encouragement. Also, to Catharina Berglund for being my most important moral and emotional support throughout this process. Dr. Dennis Bahler and Dr. James Lester helped guide and enlighten me during discussions in and out of the classroom. To Dr. David Thuente, as well, for his genuine kindness.

TABLE OF CONTENTS

LIST OF FIGURES

1 Introduction

Central to modern computing is the ability to perform logic. Indeed, logic is the framework on which the very concept of modern computation is built. In essence, computation is a careful orchestration of logical operations and functions. This logic can be simple or complex, but logic is always present at the heart of whatever computation is taking place. Computing hardware serves the purpose of facilitating this logic in a physical device. In the hardware, logical operations physically take place.

In its most fundamental form, the logic in computers is facilitated by digital logic circuits. Moreover, the basic components of these circuits are known as logic gates. Gates carry out only a single logical operation such as logical AND, OR, or NOT. It is true that these gates can constitute logic circuits all on their own. However, they are most often combined and interconnected in various ways to create more complex circuits. Now, provided that the appropriate gates and connections are chosen, it is possible to create a circuit which implements any logical function. The act of using gates and connections to build up circuits in this way is a process of careful design. Designing circuits is, naturally, an immensely important process. Without careful and correct design of digital logic, computers would not be able to function. Aside from being important, though, there is an ironic aspect of digital design. Specifically, design is both simple and difficult, simultaneously. This dichotomy is important to understand, before advancing the discussion any further.

The dichotomy inherent in designing digital logic circuits—that it is both simple and difficult—is by no means limited to the realm of computers. Rather, it is characteristic of a very important class of systems. Namely, digital design is one example of a discrete

combinatorial system [*sensu* 24,25]. The characteristics of such a system are that it has a finite collection of discrete elements which are combined to create new distinct objects. These newly created objects have different properties than the basic elements themselves. In the case of logic circuit design, gates are the discrete elements, and they are combined to create new circuits which function differently than any of the individual gates. In addition, there is no limit on the size of the combination. This is true of any discrete combinatorial system. In this case, it means that circuits may be composed of an arbitrarily large number of gates. This last fact means that there are an infinite number of new objects that can be created with a system like this. Therein lies the root of the dichotomy in digital design. First, designing new circuits is basically simple because it involves a finite number of elements that are combined in straight-forward ways. However, because combinations can be arbitrarily large, the difficult part is getting the circuit you need. More specifically, the difficulty lies in finding which of the infinite number of possible circuits is the one you want. Anyone can throw together a few gates to create a new circuit, but designing one that operates in just the right way can take considerable time and effort.

It is worth noting that discrete combinatorial systems are amongst the most interesting and useful in the world. For instance, all natural human languages are of this type. Words are combined in infinite variety to make sentences, which have a meaning all their own. Moreover, the genetic code contained in our DNA is another discrete combinatorial system. Obviously, there is much power to behold in a system of this kind. However, most of the systems we encounter in our world are not of this type. Rather, we most often come across blended systems [24,25], such as that of color. In systems like color, components are also put together to yield a new result. However, the resulting combinations have properties

which are not fully distinct from the properties of the components. Rather, the properties of the result are, in some sense, an average of the components. For instance, when two primary colors are combined, the characteristics of the new color are a mixture of the initial two. Thus, the properties that are possible to observe in a blended system are entirely limited by the properties of the component parts. This is true of most systems we encounter day-to-day, such as sound, meteorology and even cooking [24,25].

The fact that logic circuit design is a discrete combinatorial system means that the rules of design are simple, but that achieving design goals can be difficult. Fortunately, the simple rules of design are powerful enough to allow the creation of any circuit we want. However, it is the responsibility of the designer to find the circuit which best suits the design problem at hand. Conventionally, the designer of digital logic circuits is a human being. These people who are responsible for designing circuits must, obviously, have extensive knowledge of formal logic and of circuit operation. In most cases, these designers also know several established design methods which they were taught to make the process easier for people to manage. For expert designers, they may have personalized these established methods using experience and human creativity, or they may have created their own methods for design. However, the fact that creativity often enters into the design process (when humans are designing) doesn't mean that creativity is a prerequisite for the ability to design circuits of this type. Indeed, in recent years, the idea of automating the design process has spawned an interesting and fruitful field of research in computer science. In this research, circuit design is treated like any other computational problem. By automating design, the goal is to remove human effort, and human limitations, from the design process. This can be

done by taking advantage of what computers do very well, quickly examine a huge number of possible solutions.

Automation of digital design is desirable because it removes the need for human effort in deriving new circuits. However, there are other reasons which are equally compelling. One such reason, which may not be immediately obvious, is the pursuit of superior designs. As we will see, the established design methods allow human beings to arrive at circuit designs quickly and accurately. This speed and accuracy, however, is gained by limiting the realm of possible solutions. In using these methods, one is forced to substantially limit the available components, the ways those components can be configured, or some such property of the design problem. By enforcing these limitations, it is ensured that the design problem will be tractable for human beings. However, it is also highly possible that the results will be of limited optimality. As previously mentioned, some designers are experienced enough, or clever enough, to go beyond the strong limitations of the established methods. However, some number of strong limitations are always in place. In many cases, this keeps human designers from creating solutions which are optimal. By automating the design process, it is possible to take advantage of the speed with which computers can consider possible solutions. Thus, it is possible to substantially loosen the limitations, consider a wider array of possibilities, and to consequently arrive at a superior solution. Moreover, it is possible to do all this in a similar, if not better, amount of time.

1.1 Essential Concepts

Before proceeding, it is necessary to explain what is meant by ‘optimal’, in terms of circuit design. More generally, we need to define what makes one design solution better than another. Truth be told, there is no universal sense of what makes one circuit better than another. Rather, it all depends on what criteria are selected as important. However, in this paper, there is a small selection of criteria that we will be dealing with. First and foremost, is the criterion of functionality, which is of paramount importance. A circuit which always produces the correct outputs is obviously superior to one which does not. However, functionality is not always a binary property (i.e. it works perfectly, or doesn’t work at all). There are levels of functionality which can be examined, and this will become important later, for some of the new research presented in this paper. Another important circuit property that we will focus on is size. In this paper, what is meant by size is the number of gates used to construct the circuit. Obviously, smaller sizes make a circuit more efficient and this is highly desirable. Depth of a circuit will also be considered, by which is meant the longest chain of gates needed to get from the inputs of a circuit to the outputs. Other criteria, including power utilization, will be mentioned and explained in the paper, but are of lesser importance.

It is also important to note that when we are talking about computers designing circuits, we are talking specifically about designing structural representations for the circuits. That is, a representation which defines the circuit in terms of components and connections. The design specification must go beyond a simple black-box description of the circuit’s behavior, but does not need to go so far as to explain the physical properties of the circuit.

The functionality of the circuit must be derivable from the representation, in order to evaluate it. Moreover, the types of components used, and the connections between them must be included in the representation. However, exact spatial dimensions and details of operation are extraneous to the discussion presented here.

1.2 Forthcoming Topics and Sections

In automatically designing logic circuits of this type, techniques from artificial intelligence have been extremely useful. Researchers who broke ground in this area saw that logic circuit design could be formulated as a puzzle, and puzzle-solving is an area of great success within artificial intelligence. Specifically, genetic algorithms have been highly researched as a candidate for automating circuit design. Moreover, there has been a good amount of success with using these algorithms. Genetic algorithms have been able to produce better results than human designers, and in a shorter period of time. In that vain, the first half of the original research described in this paper will focus on genetic algorithms. Attempts are made to improve the performance of genetic algorithms for this purpose. Improvements are shown in both the time required to arrive at results and the quality of those results.

The second half of the research in this paper involves the application of an artificial intelligence technique that has, to our knowledge, never before been used for this purpose. Specifically, an exhaustive search algorithm is employed. The results of this research show that there are advantages and disadvantages to this approach with respect to genetic algorithms. There is a clear trade-off between the two approaches. At the same time, the

research contained herein highlights the inherent difficulty of this type of design, regardless of the approach. Nevertheless, automated logic circuit design is possible, and its results are promising.

Section 2 of this paper contains an overview of basic knowledge in logic circuit design and Section 3 contains an overview of relevant artificial intelligence techniques. In Section 4, one can find a review of concepts and relevant literature involving the application of artificial intelligence techniques to automated circuit design. Section 5 begins a discussion of new research, where a successful attempt is made to optimize genetic algorithms for automation. The discussion of new research continues in Section 6, where automated circuit design is performed by an artificial intelligence technique (exhaustive search) that is previously unutilized for this purpose. Section 7 compares the genetic algorithm approach to exhaustive search. Finally, Section 8 contains some final discussion points and suggestions for future work in this area.

2 Logic Circuit Design

For the realm of computer science, the important principles of logic were first laid out in 1854 by George Boole. His work, entitled *An Investigation of the Laws of Thought* [2], presented a two-valued algebra which constituted a formal and systematic treatment of logic [6]. Since then, this system of logic has commonly become known as Boolean algebra, in honor of his seminal efforts [12]. Boolean algebra is the fundamental framework for all logic in modern computers. As such, the ideas of this logic should be familiar to computer

scientists. Nonetheless, it serves us well to briefly review the essential ideas, but with an eye towards understanding logical circuit design.

2.1 Boolean Logic

In essence, Boolean algebra contains two types of important objects, variables and operators. From these two basic items, complex logical expressions can be built. The variables, like variables in any other mathematical system, have names associated with them (i.e. X, Y, Z, etc.). They can also take on one of two values, namely 0 and 1. In the usual interpretation, a value of 0 corresponds to a logical 'false' and a value of 1 corresponds to logical 'true'. Although it is possible to conceive of Boolean algebra with more than two values, it is irrelevant to this discussion since digital electronics deal mainly with binary values.

An operator, on the other hand, is a rule which takes a pair of variables and, based on their values, it returns a new value. For instance, the AND operator takes a pair of variables and returns 0, unless both variables have the value 1, in which case AND returns a 1. Another important operator in Boolean algebra is the OR operator. OR returns 1, unless both inputs variables are 0, in which case it returns a 0. Any operator that relates exactly two variables is known as a binary operator. Now, it is conceivable for Boolean algebra to define operators which take more than two variables as operands. However, these are not necessary and do not make the algebra any more powerful. Moreover, binary operators are simpler to deal with, so we will deal exclusively with binary operators in this discussion.

It is possible to define a whole host of possible operators in Boolean algebra. Even if we only consider binary operators, such as we are here, there are 16 possible operators that can be defined (figure 1). However, only certain operators are useful or desirable. Indeed, some are used much more commonly than others. The two operators mentioned already—the AND operator, along with the OR operator—are the two most commonly used. This is most likely the case because they were in the original conception of Boolean algebra. Additionally, they correspond very nicely with logical operators that human beings use in everyday situations. The exclusive-OR operator, or XOR, is also commonly used and will be featured prominently in discussions that follow. With XOR, a 1 is returned whenever the input variables differ in their values. Otherwise, XOR will return a 0. There are some other functions, such as NAND and NOR, which are commonly used in circuit designs, but will not be used very much in this paper. The reasons for selecting AND, OR and XOR to be our major operators in this discussion will become obvious later. Suffice it to say, however, that this selection of operators is powerful, intuitive and follows in the example of previous research that has been influential in this area.

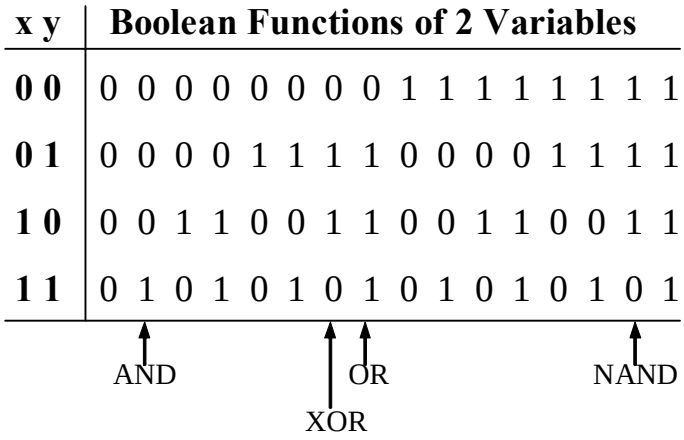

Figure 1. All possible Boolean functions of 2 variables, with some commonly used functions noted.

It should be noted, additionally, that NOT will be used as a separate operator for the discussions in this paper. NOT only takes one input variable (otherwise known as a *unary* operator) which simply returns the complementary value of the input variable. Though simple, this is a very useful operator to include. It will be symbolized in this paper with an apostrophe (as in X', the complement of X). In the truest conception of Boolean algebra, there is no need for an operator of this type because variables are all defined as having complements. The reason we include it as a separate operator is that it makes the transition to talking about real circuits easier. In most cases with circuits, complements are not simply available in the way they are thought of in Boolean algebra. Rather, they must be provided through the use of extra circuitry. Therefore, when we are designing circuits, we want to recognize this fact and make sure that it is explicit, rather than implicit, in the process.

Now, variables and operators are used to build up Boolean functions. A function is an algebraic expression that is constructed by combining some number of variables and operators, along with an equal sign [6]. On the other side of the equal sign from the variables and operators is the function name. The function can take on the value 0 or 1, similar to the variables of which it is composed. Of course, the value of the function depends entirely on the value of each variable and the configuration of the operators within the function. For instance, consider the Boolean following function, named 'F', as a simple example (n.b. 'AND' is symbolized by '·', and 'OR' by '+'):

$$F = X \cdot Y + X' \cdot Z \quad (1)$$

This function is composed of three variables: X, Y and Z. It also has four operators: two ANDs, one OR and a NOT. It can be determined that this function has a particular behavior. In other words, F is equal to 1 in certain specific instances and it is equal to 0 in others. More specifically, F equals 1 if X = 1 and Y = 1, or if X = 0 and Z = 1. Otherwise, the function equals 0.

Since this function is fairly simple, it is possible to see this behavior by simple examination. However, the most comprehensive approach to determining the behavior of this function is to write its truth table (figure 2). In other words, one can exhaustively list all the logical situations which apply to the function. To construct a truth table, one must first list which variables are present in the function. In the case of our current example function, the variables are X, Y and Z. Then, one must exhaustively list all the possible value combinations that these variables can take on. Each row represents a different combination.

X Y Z	**X'**	**X'*Z**	**X*Y**	**X*Y+X'*Z**
0 0 0	1	0	0	0
0 0 1	1	1	0	1
0 1 0	1	0	0	0
0 1 1	1	1	0	1
1 0 0	0	0	0	0
1 0 1	0	0	0	0
1 1 0	0	0	1	1
1 1 1	0	0	1	1

Figure 2. The truth table for the 3-variable Boolean function F = X*Y+X'*Z.

Next, one must determine the value of the function for each row, given the variable values that are listed there. To do this, it is easiest to simply "plug in" the variable values, apply the operators by hand, and then discover what the value of the function is in each situation. For our example function, F, we see that only

certain rows in the truth table list a 1 for F's value. If we look closer, we see that these rows correspond to situations where either X = 1 and Y = 1, or where X = 0 and Z = 1.

Just as it is possible to list the truth table for a given Boolean function, it is also possible to derive a function from a truth table. In fact, doing this is not at all complicated. The simplest method relies, primarily, on a particular type of Boolean function called a minterm. A minterm is equal to 1 in one, and only one, row of the truth table. Furthermore, a minterm can be defined by simply examining the variables which are listed in the single row of the truth table where the minterm equals 1. The minterm function is then defined as an AND-ed group of those variables (also known as a *product*), where the variables in that row with a value of 0 are complemented. Consider, again, the truth table for our example function, F (figure 2). If we wanted to make a minterm that equals 1 in the row where X = 1, Y = 1 and Z = 0, we would define the minterm thusly:

$$m_6 = X \cdot Y \cdot Z' \quad (2)$$

Note that X and Y are in their true (uncomplemented) form, but Z is complemented. This corresponds to the values of the variables in the specified row. Furthermore, this function, m_6, is equal to 1 in only the specified row. Also, note that the name of the minterm, m_6, indicates that the only 1 is located in the 6^{th} row of the truth table. This last item is by convention.

If we understand how to construct minterm functions, it is easy to then use the minterms to construct a more complex function. First, a minterm must be constructed for each 1 located in the rightmost column of the truth table. In other words, there must be one

minterm function for each situation in which the function equals 1. Second, a new function must be constructed which simply ORs (or sums) the minterms together.

It is beneficial, of course, to go through a quick example. Going back again to our example function F (figure 2), one can see that there are four rows in which the function equals 1. The following minterm functions can be derived from those four rows, in exactly the same way as before:

$$m_1 = X'\cdot Y'\cdot Z;\ M_3 = X'\cdot Y\cdot Z;\ M_6 = X\cdot Y\cdot Z';\ M_7 = X\cdot Y\cdot Z \quad (3)$$

Once these functions have been constructed, the next step is to combine them. They will be summed together to form a new function. The function is as follows:

$$F = X'\cdot Y'\cdot Z + X'\cdot Y\cdot Z + X\cdot Y\cdot Z' + X\cdot Y\cdot Z \quad (4)$$

This function will show the exact same behavior as described in the truth table for function F (figure 2). Moreover, it will also show the same behavior as the previous function given for F. Indeed, the two functions are equivalent, even though the forms are different. The latter, longer form of F can be thought of as a complete listing of where the function is equal to 1. However, the complete listing is not necessary to specify the target function. Rather, some amount of condensing can take place. The prior form of F represents such a condensed version.

2.2 Logical Forms and Simplification Techniques

The different forms of F that are given, above, are examples of two very important forms. The latter, wherein all the minterms are listed in the function, is known as a canonical form. Canonical forms often contain more operators than other formulations, but they also require the least amount of effort to arrive at. Indeed, we have just seen how simple it is to construct a function in canonical form. There are other types of canonical forms, as well, which depend on a complete listing of rows that are equal to 0 (known as maxterms). These canonical forms are equally easy to construct. However, canonical forms almost always require more than the minimum number of operators.

The prior form of F is a type of standard form. Functions in that standard form are still composed of AND terms connected by ORs, but require fewer operators. This form is also called sum-of-products. Other standard forms are composed of OR terms connected by ANDs (product-of-sums), while still requiring fewer operators. One can think of functions in standard forms as being similar to those in canonical form, only they are condensed.

It is usually desirable, as one might expect, to have a function with fewer operators. Hence, standard forms are usually desirable, since they are smaller. Remember that the first form of our example function, F, was a type of standard form. Moreover, this form of F required only 4 operators (2 ANDs, 1 OR and 1 NOT), whereas the canonical form requires 15 (8 ANDs, 3 ORs and 4 NOTs). That is quite a substantial savings in the number of operators. Fortunately, the route to constructing a function in standard form is fairly straight-forward. There are several established methods which make it easy for human beings to put

functions into standard forms. These methods are commonly known as logic simplification techniques, for obvious reasons.

These methods usually focus on convenient ways to group minterms together into smaller expressions. It is possible that several minterms may be covered by a much smaller term and will not need to be listed, exhaustively, in the function. To borrow an example from our function F once again, we can consider the minterms listed in the canonical form. Remember, all the while, that the terms are OR-ed together in this canonical form of the function. Therefore, if any term is equal to 1, the whole function is equal to 1. Specifically, the first two minterms listed are X'·Y'·Z and X'·Y·Z. Notice that both terms contain X in complement form and Z in true form. The variable Y, on the other hand, is seen variously in true and complement form. Logically, we can take this to mean that if X = 0 and Z = 1, the value of Y doesn't matter. Either way, one of the terms will be true, and consequently the whole function will be true. Therefore, we can say that the entire function will be true if X = 0 and Z = 1, without consideration for Y. So, we replace the first two minterms with, simply, X'·Z. Doing so does not change the behavior of the function at all. The first two minterms of the function are "covered" by the shorter term.

This type of condensing is exactly what the simplification techniques aim to do. Of course, they formulate the idea of condensing in a much more convenient way. With the methods, the choices for which minterms to condense are made more obvious. One method, called the Karnaugh Map method, presents the minterms as an array of 1s and 0s. The goal is to draw the largest boxes possible around groups of 1s or 0s. Once this is done, the term which replaces them is simply a grouping of their common variables [6]. Another method,

called Quine-McClusky for its creators, avoids the map representation in favor of an ordered listing of the differences between minterms.

Sometimes these methods are called minimization techniques, implying that they generate functions which use the fewest number of operators. However, this is a bit of a misnomer. There is no guarantee that a function constructed through these methods is in its smallest form. The trouble, however, is not with the methods, but with the form itself. Indeed, there is no guarantee that the standard form of any function is the smallest possible form. Despite this, standard forms are extremely important for two main reasons. First, they are easily obtained by certain methods, for example, those mentioned above. Second, they are smaller than the canonical form. In some instances, they are substantially smaller, as was the case with our example function, F. One problem, however, is that standard forms are just that, standard. By definition, they always have the same basic properties. The functions that result from using the methods above are either in sum-of-products or product-of-sums form. However, we know from our earlier discussion that many, many possible combinations exist, and many of them are not in standard form. Although this standardization makes the functions easy to produce, at the same time, it sometimes makes them limited in terms of minimization. Although it is possible for a function in standard form to utilize the absolute minimum number of operators, in many instances this is not the case. If one is open to arranging operators in very unconventional ways, there may be other functions which behave identically while making use of even fewer operators.

2.3 Logic Circuits

Now, the reason why all this information about Boolean functions is relevant here, is that the digital logic of computers is simply an implementation of the ideas from Boolean logic. Indeed, the variables of Boolean functions are simply the same as inputs to logic circuits. Similarly, the logical operators in Boolean functions are implemented with logic gates in circuitry. Refer to figure 3 for an illustration of logical operators and their corresponding logic gate schematic symbols. Just as in Boolean algebra, where variables and operator are combined in endless ways to create new functions, so are inputs and gates combined and connected in different ways to create new circuits. The output of a logic circuit is determined by the behavior of the function it implements.

Logical Operation	Schematic Symbol
AND	
OR	
XOR	
NOT	

Figure 3. The names of logical operators that are important to this work paired with their schematic symbols.

Hence, the discussions of logic simplification methods, and the limitations thereof, also applies to the design of logic circuits. Rather, the ability of simplification methods to cut the number of logical operations is even more crucial when we talk about implementing circuits. The reason is simple, every logical operation is another piece of physical circuitry

that is necessary to include. In order to not waste materials, and to make our circuits more efficient, we must ensure that our circuits are as small as possible.

The difficulty of exploring a larger diversity of functional configurations, however, is that the number of configurations quickly balloons. As we relax the limitations on the types of configurations we will consider, the numbers begin to mount. Furthermore, if we then try to consider circuits with more inputs and a broader variety of gates, the numbers climb even faster. It soon becomes impossible to think of a human being doing the designing. There are simply too many combinations to consider. Hence, it is desirable to employ computers for the task of circuit design. Surely, they can analyze possible solutions many times faster than human beings can. In doing so, they may be able to find circuit designs which are more efficient than any design a human could devise; at least, more efficient than a human could devise in a reasonable span of time.

As we begin to talk about automating the design process, our main concerns change. No longer are we worried about analyzing the logic behind the circuits, or about coming up with good designs on our own. These concerns should be passed on to the computer for consideration. Mainly, we should be concerned with properly formulating the problem so that it can be automated. Also, we should be concerned with finding the correct automation process. That is, a process that produces agreeable results, and that is reasonably efficient. An exploration of these new concerns has already begun in the literature. Surprisingly, perhaps, many of the concerns are being put to rest by using ideas from artificial intelligence.

3 Search Techniques from Artificial Intelligence

Artificial intelligence techniques have been crucial in automating logic circuit design. Specifically, the idea of search algorithms has led to some important gains in this type of automation. Before it is possible to fully appreciate the reasons for this, however, it is important to review the essential aspects of artificial intelligence search techniques. The discussion presented here, similar to that given in Section 2, is not intended to be comprehensive. Rather, it is a review of major concepts and certain specific details, given with an eye towards automating logic circuit design. The research which will be presented in this paper follows two distinct approaches. These two approaches are distinguished by the search techniques from which they are built. Namely, they are based on types of systematic search and local search. As such, the following review of search concepts will begin with systematic search, and proceed to local search. Afterwards, a clearer connection to logic circuit design can be made.

3.1 Search Basics

Solving problems by searching is a powerful and useful idea. It is a way to get to a solution by examining different sequences of actions, when there is no single action which solves the problem [28]. Many problems can be formulated in a way which agrees with the search paradigm. To do so, one must first formulate a representation for the state of the problem as it currently is. This is known as the initial state. There must also be the concept of a goal state, what the problem looks like when it is solved. Next, one must determine a

series of legal actions which are to be carried out on the initial state. By performing the legal actions, it is possible to create successors of the initial state. Legal actions may then be performed, in turn, on the successor states. Hopefully, some actions will lead towards the goal. Thus, one must have a test which determines if any state represents the desired goal. This is known as the goal test. At each step, when a state is selected to have an action performed on it, the goal test is applied to make sure it isn't already a goal state. If it is, then the sequence of actions that was used to get from the initial state to the goal state represents a solution to the problem.

Often, there are multiple ways of getting from the initial state to a goal. In these cases, one solution can be preferable over another. It is possible to determine quality of solutions by considering the number of actions required to get from the initial state to the goal. Additionally, different actions may have different costs associated with them. If this is the case, then the costs can be summed, rather than just considering the number of actions.

The name "search" refers to this technique because it is applying many combinations of actions to the initial state, in *search* of an acceptable combination. An acceptable combination is one which constitutes a solution, and which is low enough in cost to satisfy some specified criteria. Once an acceptable combination is discovered, one can get to the goal state by applying the sequence of actions that are specified in the solution.

For an example of problem-solving via search, consider the classic route-finding problem [28]. Specifically, assume we are in New York City, and we want to travel by car to San Francisco. That is, our initial state is us being located in New York. Our goal is to be located in San Francisco. Our goal test, for lack of a global positioning system, will be to scan the horizon for the Golden Gate Bridge. The problem, obviously, is finding a good

route to travel. Which cities should we travel through on the way to our destination on the west coast? At each step, the legal actions are to drive to any of the nearby cities on a highway. We may chose to drive south first, to Philadelphia and then on to Washington. On the other hand, we could go west to Buffalo and then continue to Detroit. There are many possible paths to choose from, any one of which may constitute an acceptable solution. Note, too, the distinction between the goal and the solution. The goal is to be in San Francisco. The solution, on the other hand, is an acceptable path to get from here to there. Once we have a solution (i.e. a path), we can apply the actions that make up the solution to achieve our goal (i.e. to get where we are going).

Before continuing, there is some terminology which must be explained. Search is often thought of as building a tree structure which is called a *search tree*. For instance, the tree which is formed by recording all highway paths as one leaves New York is a search tree. The nodes of the tree are called *search nodes*. Each node in the tree corresponds to some state of the problem, such as being in New York. The root of the search tree is the initial state, and when the successors of that node are generated through the application of legal actions, the search tree begins to grow. Generating all of the successors of a node, through the application of all legal actions, is known as *expanding* that node. In practice, the successors of a node are always generated all at once, rather than at separate times. Therefore, expanding is an important concept. Now, any node which is in the tree, but has not been expanded, is known as a *fringe node*. The set of all fringe nodes is, intuitively, known as the *fringe* of the tree. At the beginning of any search, the fringe is composed entirely of the initial state. The first step is always to expand the initial state, and then all of

its successors become the fringe. Once it has been expanded, the initial state, or any other expanded node, is no longer a part of the fringe.

The route-finding problem offers us a fine example to explain this terminology. The root of the search tree is the initial state, namely the state of being in New York. Expanding the root node creates several new nodes, all of which branch out from the root. In expanding, we apply all legal operators for driving out of New York. Legal operators in this case may be driving South, North and West. Note that driving east may be a sensible operator for the problem as a whole, but it is not a legal operator for the case of driving from New York (unless your car is amphibious!). The new nodes which are generated might represent the state of being in cities near to New York, such as Philadelphia, Buffalo, etc. These new nodes also compose the fringe of the search tree, because they have not been expanded themselves. As we expand nodes, the paths we are searching become longer and longer. Hopefully, we will eventually reach our goal of being in San Francisco.

3.2 Systematic Search

Different search algorithms are distinguished by the order in which they choose to expand the fringe nodes [28]. By choosing different orders of expansion, some may find answers more quickly, and some may find answers that are superior. The search paradigm is very broad, and there are many variations on the fundamental ideas. Some of the oldest and most widely used variations fall under a broad sub-category which can be called systematic search. Search techniques in this category are systematic, not because they leave no stones unturned, so to speak. Although, that is certainly an option. Rather, they are systematic

because they keep track of which search options have been explored and which haven't [28]. This is done by keeping the entire search tree in memory as it develops. In this way, they can ensure that all search options which are deemed relevant can be explored. There are alternatives to systematic search, and they will be seen later.

Systematic search is, perhaps, best exemplified by its simplest and most intuitive manifestation. Specifically, the type of search known as breadth-first search (BFS). It illustrates the ideas of systematic search very nicely, including all of its strengths and weaknesses. BFS explores all possible options at each step, until a solution is found. What this means, in practice, is that the initial state is expanded, followed by all the successors of the initial state, then all the successors of those successors, and so on until a solution is found. In other words, each level of the tree is explored completely before heading further down the tree.

Not only is BFS systematic, but it is exhaustive. It considers all possible search paths at each step. Therefore, it will always find a solution if one exists. This is a very desirable property. Moreover, by considering nodes closer to the root first, BFS will always find the solution which is shallowest in the tree. In most cases, this shallow node will also represent an optimal solution. In fact, if the problem is properly formulated, one can be guaranteed that BFS will find the optimal solution. Again, this is a highly desirable quality for a search algorithm to have. The downside of BFS is that for many problems, its time and memory requirements are highly impractical. Not only does each fringe node need to be considered at each step, but each node and its successors all need to be kept in memory as the search carries on. As a result, both time and memory requirements grow exponentially with the depth of the tree.

There are ways to cut down the costs associated with an exhaustive search method such as BFS. One such way is with the use of heuristics. A heuristic is a problem-specific function which serves an important purpose, it estimates the cost of getting from any node to the goal node. With the information provided by heuristics, a search algorithm can determine which nodes may be closer to the goal, and thus make more informed decisions about the best node to expand next. By choosing more wisely from the fringe, an algorithm can potentially avoid expanding many nodes which lead down undesirable paths. It is important to emphasize that heuristics can only estimate, though. If one could always know the exact distance, then the problem would essentially be solved and searching would be completely trivial.

For our route-finding problem from New York to San Francisco, a good heuristic might be to take the straight-line distance from each location to San Francisco. This distance stands as an estimate of the highway distance which we would travel on the actual drive. However, the estimate is probably pretty accurate, to within a certain degree. If we actually knew the highway distance from any location to San Francisco, the problem of finding a good route would be trivial. Thus, an estimate will suffice.

Some heuristics can produce estimates which are highly accurate. If an accurate heuristic is used, then the algorithm is more likely to expand nodes that are relevant to good solutions. Nodes which lead to bad solutions, or no solutions, can be avoided altogether. The savings in time and space for search algorithms can be substantial. Moreover, if heuristics are used carefully, one can still be guaranteed that the solutions will be optimal.

The most immediate problem with heuristics, however, is that they can be quite difficult to design. They require specialized, in-depth knowledge of the problem at hand.

This can take time and a great deal of effort. For some problems, it may be nearly impossible to accurately estimate the cost to the goal. There is a more general problem, as well. Even with excellent heuristics, systematic search methods can still utilize large quantities of space in memory and take a long time. To some extent, this is the price to be paid for the benefits and guarantees of a search which is systematic.

3.3 Local Search

There are alternatives to systematic search. Some search algorithms take a completely different approach to the concept of search. Indeed, they are intended to work on a completely different formulation of problems. For formulation of some problems, the path to the goal doesn't matter. All that matters is being at the goal. In these situations, it is not necessary to keep track of the path from the initial state to the goal state. Rather, it is sufficient to keep track of only a single state. This state, known as the current state, represents the situation as it currently stands. At each step, we can apply some operators to the current state to alter it, just as before. The main difference here, though, is that once the operator has been applied to the current state, the new state replaces the old state. There is no keeping track of the global picture, where the search has been or what decisions were made. Hence, this type of search is known as local—as opposed to systematic, or global—search.

To clarify local search, we will follow an example, fairly closely, from Pfeifer and Scheier's book *Understanding Intelligence* [26]. Suppose we are given a string of 10 letters that is completely random. For the sake of explanation, say the randomly generated sequence of letters is "wlmldtjbkp". The goal is to make this string spell the target word "university".

However, we don't know the target string. Instead, all we have is an evaluation function which, when presented with a string, can tell us how many correct letters are in their correct positions. So, we begin searching in a simple fashion. At each step, we produce five copy strings, and randomly change one letter in each. Each string is then fed into the evaluation function and rated based on the letters it shares with the target. The more letters it shares with the target, the better. The best string of the five then replaces the initial string, and the other four copy strings are erased. Eventually, by selecting the best string at each step, we will arrive at a string that matches completely. Note that there is no record of the search procedures that takes place. During each step, the memory requirements never exceed that of 60 letters. The requirements will never grow, either, as the algorithm proceeds. Also note that, even if we kept the records around, they would be virtually useless once an answer was found. We don't care how we got to a matching string, in this case. All we care about is that we did match the target string. This type of local search algorithm is known as hill-climbing because it always moves in the direction of increasing or equal value [28]. It is, perhaps, the simplest local search algorithm, but it is also quite effective.

Local search algorithms vary in how quickly they can converge on solutions. Many different algorithms exist, and each is different. However, even simple algorithms, like the one presented above, can find solutions after only a small number of steps. Small, that is, relative to the 26^{10} possible letter arrangements in a 10-letter string. Also, local search algorithms offer huge savings on space requirements, especially as compared to systematic search algorithms. The space requirements are usually constant throughout the algorithm's runtime.

On the other hand, erasing the records of a search has its detriments. Some common local search algorithms cannot even guarantee that they will find solutions. This is because, without records of past search activities, it is impossible for an algorithm to reconsider old options. Without this ability, the algorithm cannot tell if it is stuck in a local maximum, or some such sticking point in the search space [28]. In essence, this is the downside of removing the systematic element of searching. Although, local search often does find solutions. There are many ways to alter the simple algorithms so that they are much less prone to getting stuck. A related problem is that these algorithms often cannot guarantee any level of quality in solutions. The algorithm might find one solution, but could get stuck and never discover that there is an even better one. Despite these problems, there continues to be much attention on local search algorithms because of their benefits.

One type of local search algorithm which has attracted much interest is genetic algorithms (GAs). These algorithms attempt to capture the power of one of biology's most powerful concepts: natural selection [14]. GAs were first proposed in 1975 by John Holland, in his work entitled *Adaptation in Artificial and Natural Systems* [9]. Like our hill-climbing example, GAs have space requirements that are small and have a fixed ceiling. However, GAs are not so drastic in cutting the amount of nodes they keep around. Rather than one current node, there can be many, perhaps even hundreds. The number is finite, though, and usually still small with respect to the entire search space. In GA terminology, this finite set of nodes is known as a *population*. Each member of the population is known as an *individual*. As above, an individual is commonly represented as a string of numbers or characters. Even if the problem doesn't involve strings in the solution at all, the different problem states are still converted into string representations. This is meant to be similar to

the method of encoding in DNA, and how protein sequences are encoded with a four-letter alphabet. At each step, an entirely new set of individuals will be constructed from the individuals in the current set. All the individuals in a set at one time are, together, called a *generation*. It is the method of producing a new generation that makes GAs so unique. The method attempts to simulate sexual reproduction.

Going back to our string-generation example from above, let us formulate the problem for a GA. Instead of one random string at the beginning, we start with some finite number larger than one. Say it is 50. To begin the construction of a new generation, the GA first evaluates each of the 50 strings by submitting them, as before, to the evaluation function. In the case of GAs, the evaluation function is known as the *fitness* function, since it fits with the biological terminology. The more fit an individual is, the better. For the string-generation problem, the fittest possible individual would be the string "university".

Once the fitness scores are obtained for all 50 strings, pairs are chosen from the among the 50 for reproduction. Each pair will yield a new pair of strings via reproduction, so we choose 25 pairs to keep the population size the same. The pairs are chosen randomly, but are weighted with respect to the fitness scores. Thus, the pairs are largely composed of individuals with higher fitness scores. Some extremely fit individuals may be chosen several times for pairing. Also, lower-scoring individuals always have some chance of being chosen. The reality, though, is that extremely low-scoring individuals may never be selected at all for any pair. Again, all this is intended to simulate the biological situation in which the more fit organisms pair off for reproduction.

During the reproduction phase, each pair is subject to *crossover* (figure 4). During crossover, a point is chosen at the same position in both strings in a pair. Both strings are

then spliced at that position and the pieces after that point are exchanged between strings. For instance, if the crossover point is at the 3rd position in the pair, then the 4th through the 10th letters of the first string will be exchanged with the 4th through the 10th letters of the second string. Finally, each position in each of the strings has an independent probability of being mutated. That is, the letter will be randomly changed into a new letter. The mutation probability is predetermined and is usually fairly small. One additional operation that can take place during reproduction is *elitism*. This means that the best string from one generation will always be carried over, unchanged, into the next generation. Though this idea is questionably biological, it is a practical consideration. It prevents the best idea yet from simply vanishing in the coming generation. This, despite the fact that searching is still proceeding.

Crossover is the operation that makes GAs truly unique. Combining crossover with the more basic idea of random mutation is also unique, and also powerful. Mutation allows for exploration of random parts of the search space [28], while crossover allows trading and exchanging of ideas from different strings. This combination seems to be extremely effective in finding solutions. Indeed, they have

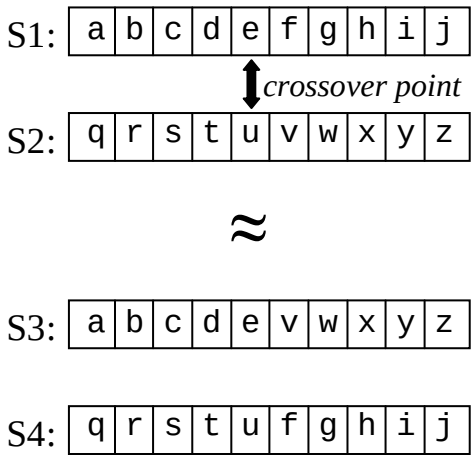

Figure 4. During crossover, strings (S1 and S2) are spliced and recombined to create new strings (S3 and S4) around the crossover point.

had substantial impacts on many types of problems. Aside from their effectiveness, they also share many of the general benefits of local search, most notably the reduced space requirements. However, there is still a lot that is unknown about how effectively GAs operate in general and why it is that GAs seem to work so well. They also suffer from the problems of the local search paradigm. Specifically, they are not systematic or exhaustive, and they make very few guarantees about the quality of solutions they can produce. Nonetheless, they continue to capture the interest of many researchers for their ability to successfully negotiate a wide variety of problems.

4 Applying Search Techniques to Logic Circuit Design

Automating logic circuit design is both desirable and difficult. The design process can involve sifting through an immense number of gate configurations to find the optimal one. Fortunately, search techniques were designed for sifting through immense numbers of configurations. It is an obvious progression to consider search for finding combinations of logic gates. Search has been used to successfully solve such diverse problems as robot navigation [30,31,32,33], puzzle solving [28] and DNA strand alignment [5]. It has been used to find desirable and winning combinations on chess boards. With the proper formulation, the problem of designing logic circuits does not look dissimilar to any of the problems for which search is normally used.

Formulating the problem for search is fairly intuitive. Gates and the connections between them are taken as the primitive search elements. In other words, the search proceeds by combining and rearranging gates and connections. There is some selection of gates which

is predetermined and available for inclusion in the circuit. Each type of gate can be used in the design, or not. For the sake of simplicity, only two-inputs gates are usually considered. Each individual gate gets its inputs from either the circuit inputs or from other gates. The outputs from each gate can either connect to the inputs of another gate, or they can contribute to the overall output of the circuit. Usually the circuit's overall output comes from some single, designated gate. The overall number of used gates can grow or shrink during the search.

The primary goal for this type of search is, obviously, to produce circuits which are fully functional. That is, the circuits must produce the correct outputs for each of the possible inputs. The functionality of a certain configuration might be tested by building a physical circuit and testing it. Alternatively, one can choose to simulate the circuit's operation. Such a simulation entails propagating the logical values of the inputs through the circuit. Beginning with the inputs, the values proceed through the gates layer by layer, just as they would in a physical circuit. At each gate, the appropriate logical function is applied. When all the values have been propagated, the value of the designated output gate is read. This must be done for every possible combination of the input values. If all the output values match their respective input combinations, then the circuit is fully functional. Functionality is not the only concern, but it is always the most important concern. Other secondary criteria may be part of the search as well, such as size, depth, power dissipation and gate selection. Information about these criteria are easily obtained by examining the circuit. For instance, the size of the circuit can be determined by simply counting the number of gates which are used.

4.1 Genetic Algorithms for Logic Circuit Design

Up to the present, most research has focused on using local search algorithms for the design of logic circuits. More specifically, genetic algorithms have been the most common choice. One reason why local search has been preferred is that the search space for logic design is enormous. Indeed, we will see later that search space grows exponentially with the number of gates required. It is so large that space-saving nature of local search seems to be a necessity. Another reason why local search has been preferred is that only the end-product of the search seems to matter. As we saw before, local search does not remember the path from the initial state to the goal state. Therefore, it is suited for problems in which the path to the goal state is irrelevant. In other words, the goal state must constitute a solution all by itself. It seems intuitive that logic circuit design fits this mould. It should not matter how the search progresses, as long as the good circuit is produced in the end. Therefore, it seems pointless to use a systematic search technique which remembers the entire search path. Not only is the space required for that information huge, but it is not useful anyways. We will see later that the path information becomes relevant if the problem is re-formulated. However, with the more obvious formulation, local search seems to be best choice.

GAs are the most common search algorithms used for automating logic circuit design. The research to date has favored their use overwhelmingly. Often, this has lead to great success as GAs seem well-suited for this purpose. In order to use GAs for this purpose, though, there must be some additional formulation of the problem. As we have seen, GAs use strings as their basic elements, in the same way that biological systems use DNA strands. Therefore, if we are to use GAs for circuit design, all of the information about gates and

connections must be encoded in a string. In accordance with the terminology from biology, this string is known as the "genotype". The genotype is an encoding of all the relevant information about the circuit. The relevant information which is encoded is known as the "phenotype". The phenotype includes the gates used in the circuit, the connections between gates and other essential properties. The phenotype can be derived from the genotype, and in turn, the operation of the circuit can be derived from the phenotype. The way in which the genotype encodes the phenotype varies, since there is no standard method for formulating the encoding. There must be an encoding, however, since it is essential to the concept of GAs. The specific encoding used in this paper will be seen later.

4.2 Two GA Camps

From the early investigations, the application of GAs to circuit design has been divided into two camps. These two camps differ, mainly, in the way they implement and test the potential circuits. During the search process, every circuit which is a potential goal circuit must be tested for functionality. This can be done by either physically implementing the circuit and testing it, or it can be done by simulating the circuit's operation [18]. Therein lies the division in this research area. If the GA tests the circuits by first implementing them in hardware, it is said to be performing *intrinsic* evolution. GAs which simulate their potential circuits are said to perform *extrinsic* evolution. Both methods have inherent advantages and disadvantages. It can be difficult to figure out whether hardware or simulation is superior. Indeed, this issue has arisen in other fields of research, and it has

proven equally difficult to resolve. Most notably, there is much debate in the literature of experimental robotics over simulation versus hardware embodiment [37].

The debate over this issue revolves around the long-fought trade-off between getting solutions of better quality and getting solutions more quickly. On the one hand, simulation allows for rapid implementation of circuits. The parameters of the circuits, and of the problem in general can be changed with ease and expediency. The components being utilized can be changed, altered, or even invented without much difficulty. Also, analysis of the circuits is easier since simulations track all of the properties of the circuits by their very nature. For the researcher, this ultimately means more design freedom, as compared to the intrinsic approach [18]. However, the extrinsic approach does limit the overall realm of possible outcomes. By using simulations to evaluate circuit designs, the researcher is necessarily making some simplifying assumptions about the testing environment. In the process, this is essentially circumscribing exactly which circuits will work. It eliminates the possibility that something unexpected might happen, that there is some subtle effect in the design which causes a seemingly useless circuit to function properly [18]. As a result, it is possible that extrinsic evolution will miss some extremely novel design. However, it is possible that intrinsic evolution will produce such novel designs.

4.3 Intrinsic Evolution

Implementing the circuits in hardware may seem prohibitively laborious and time-consuming, unless one recognized the benefits of using Field Programmable Gate Arrays (FPGAs). An FPGA is essentially an array of unassigned logic blocks, with unassigned

connections between them [29]. The logic blocks can be programmed to function as logic gates. The connections may be programmed to take on different configurations. What this means is that an FPGA can be configured to function like any number of digital logic circuits. Moreover, this can be accomplished with great speed. Therefore, it is feasible that the circuits considered by a GA can be implemented in FPGAs and tested as real, physical circuits.

The research into intrinsic evolution was thought to have distinct advantages from the early stages. The early work by Thompson [30,31,32,33] attempted to evolve functional circuits intrinsically on an FPGA. The efforts were successful and also showed that extremely novel designs were indeed possible through this approach. Circuit configurations were developed which were quite distinct from human designs. Most notably, it was possible to evolve designs which were characterized by a distinct lack of temporal coordination, the possibility for which would not have existed with a standard simulation [7,30,31,32,33]. Unfortunately, the final designs suffered from several problems. As expected for intrinsic evolution, the analysis of circuit operation proved rather difficult. Unexpected problems also arose from this method. For instance, the novel designs often relied on particular idiosyncrasies of the particular FPGA being used. Therefore, the designs were often not functional when transferred to other FPGAs. Additionally, the designs usually operated, to some extent, in the analog domain. Even though they evolved on the logic blocks of an FPGA, they were not fully digital. This last item is not inherently problematic, but it does make the designs extremely sensitive to voltage and temperature [19], which is another reason why circuits designed in this way are extremely difficult to reproduce.

Work by other researchers attempted to apply this technique to a variety of circuits [7,21,30,31,32,33,34]. By and large, they suffered from similar problems; they were difficult to analyze or were not reproducible. Attempts were made to overcome the latter problem [3,19]. This mainly involved the recognition and purposeful elimination of designs which relied on highly asynchronous signals, analog operation or on idiosyncratic properties of individual FPGAs. Consequently, this also removed all methods for producing novel designs other than strict reconfigurations of components. While this could still produce designs beyond what human designers could do, it restricted intrinsic evolution to the point where it could not produce designs that were any more novel than those produced by extrinsic evolution. In other words, it eliminated the main advantage of intrinsic evolution over the extrinsic version.

4.4 Extrinsic Evolution

As a consequence of the troubles with intrinsic evolution, the extrinsic camp has been favored in recent years. However, the research on this did not begun only recently. Rather, it began even earlier than research on the intrinsic methods. Some of the earliest work of was that of Koza [1,14,15,16], which was both foundational and influential. His work focused mainly on analog circuit design, but also touched on using GAs to develop Boolean expressions and logic circuits. The work of Lang [17] further developed the application of GAs to logic circuits. The focus for this early research was on evolving circuits which were functional, but not necessarily optimal according to any other criteria.

An important question from the beginning was how to formulate a logic circuit in a suitable way for the search process. That is to say, what should the phenotype of the circuit look like? This decision determines much about the range of possible circuits and about the computational requirements of the search. For Koza and Lang [1,14,15,16,17], circuits were formulated as trees of gates, which was powerful in terms of the types of circuits it allowed. It essentially allowed any combination of any size to be represented. However, with that power came large demands for memory and computation time. The tree formulation was also shown to produce circuits which were highly redundant [4].

Attempts were made to explore other circuit formulations [11], but the one which has gained the most favor is the array formulation. In this formulation, a circuit is conceptualized as an array of logic gates and connections between them (figure 5). This is conceptually similar to the way an FPGA is structured. At one end of the array are presented the inputs to the circuit, and at the other end of the array are the outputs. Each gate at a particular location is a member of an array column, and it can get its inputs from any gates in the previous column. The gates in the left-most column get their inputs from any of the circuit inputs, rather than any gates. This formulation was

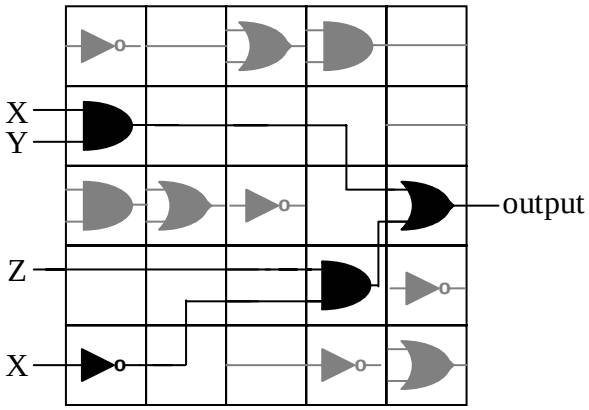

Figure 5. One example of the "array formulation" for logic circuits, which is utilized by the genetic algorithm in this paper as a search node.

developed first by Miller [22,23]. It serves the important purpose of preserving much of the power of Koza's tree formulation [1,14,15,16], in that a large number of configurations are possible. But by loosely limiting the size and the possible connection schemes, it cuts back on the resources needed to support that formulation [4].

Miller [22,23] also intended this formulation to have another important benefit. He used it to optimize his evolved circuits for size. Circuits would begin by evolving in an array of some size. If they could be successfully evolved, then evolutionary process would be repeated again with a smaller array. This would repeat until a circuit could no longer be evolved in reasonable time due to the array being too small. The circuit which evolved in the smallest possible array was considered to be the optimal one [22,23].

The array formulation was influential in the work of many researchers who followed, including the work in this paper. Kalganova [13] altered the array formulation to allow larger circuits to be evolved. This was done by allowing the array positions to take on the functionality of small circuits, as well as individual gates. Larger circuits could be evolved this way, but it is unclear whether these designs were better than conventional designs. Later, Vassilev [35,36] used GAs and the array formulation to optimize circuits automatically. He represented conventional, already functional designs in the array and evolved them towards optimality with respect to the number of used gates.

An important step was taken when Coello [4] put together previous ideas from circuit evolution research and created an algorithm that seamlessly and efficiently automated the evolution of both functionality and optimality. This algorithm also used the array formulation. More importantly, it conceptualized the evolution of functionality and optimality as back-to-back stages. First, circuits would be brought up to fully functional

status and then they would be judged on the basis of their other properties. This allowed the evolution of functionality to take precedence, while not dominating any other criteria. This research of Coello [4] serves as the primary basis for much of the research in this paper. That research is the starting point for our attempts to optimize GAs for logic circuit design. As such, there will be more detailed discussion of the work later on, in direct connection with this research.

The benefits of applying GAs to logic circuit design have been as good as expected. By automating the entire process, GAs have been able to quickly develop circuits which are fully functional. Moreover, some circuits which have been developed are superior to those designed by humans [4]. Of course, there is no guarantee that this type of algorithm will return a solution which is better than human designs. Nor is there any guarantee that GAs will return the optimal answer to a design problem. Most of the time, GAs can be trusted to return very good solutions to a logic circuit design problem. However, in order to get guarantees about solution quality, one must turn to systematic search.

4.5 Systematic Search for Logic Circuit Design

Depending on the exact algorithm chosen, it is possible for systematic search algorithms to return optimal solutions every time. As we have seen how BFS can exhaustively examine all possible combinations in order to find the optimal solution. In terms of logic circuit design, a search like this would have to examine every possible combination of gates, from one gate up to some large number, until a functional circuit is discovered. If the search proceeded from the smallest number of gates to the largest, the first

functional circuit encountered would be guaranteed to be the optimal one with respect to size. However, there is no literature on applying systematic search techniques to this problem. This, despite the fact that solution-quality guarantees are a good thing, and the process seems simple.

There are reasons, though, why some may have thought that looking into this was a bad idea. It is unclear, though, whether those reasons are valid. One reason is that we know search algorithms like BFS can have prohibitively large requirements for memory space and computation time. Since logic circuit design is a problem with a huge number of possible combinations to examine, one might reason that the requirements of the algorithm would quickly make it unfeasible to use. Another reason why systematic search has never been investigated for logic circuit design is that the problem formulation is rather odd. In the intuitive formulation of logic circuit design for search, it would appear that the end-product of the search (i.e. a functional circuit) is all that matters. As such, local search seems to be the obvious choice for solving this problem. However, the intuitive formulation of the problem is not the only possible one. Logic circuit design can be formulated for systematic search, even if it seems strange. By doing this, it will be possible to see whether systematic search is actually as impractical for this purpose as it may seem. If it is not so impractical, we will also be able to see what optimal solutions look like for various circuits of our choosing. In any event, we will be seeing it for the first time.

5 Optimization of Genetic Algorithms

The first part of the research in this paper optimizes GAs for extrinsically designing logic circuits. The conceptual groundwork which we follow was laid by Coello [4]. In that research, GAs were used in an attempt to design circuits which were fully functional and optimized for size. The algorithm was able to design circuits which, in several cases, were superior to those designed by human beings. That is, they used a smaller number of gates while maintaining full functionality. Thus, the work done by Coello [4] was highly successful in its pursuits.

One of the most significant contributions of Coello [4] was the seamless integration of optimization with functionality. The algorithm went beyond selecting circuits which were simply functional, to select based on circuit size, as well. Using GAs to search on multiple criteria in this way can be nuanced and difficult. It must be approached with extreme care, since the different criteria may have distinct levels of importance. This is certainly the case with logic circuit design, where functionality is clearly the most important criterion.

Searching for multiple criteria relies on changing the fitness function, since it is what all circuit selections are based upon. For instance, suppose that two evaluations are calculated for each circuit based on its size and functionality. Then, the fitness function could simply be the sum of these scores. That way, a change in either criterion would alter the overall fitness of the circuit. However, in practice, this simple scheme is not very effective. One criterion tends to beat out all others, causing virtually no optimization to take place on the additional criteria. Some work has attempted to overcome this problem by

borrowing specific ideas from biology, such as Pareto domination to ensure that all criteria are emphasized equally [10].

These schemes do not fit with circuit design, though. Designing logic circuits is a problem with special requirements. Specifically, functionality must remain the criterion that trumps all other criteria, but it must not drown out the possibility for optimizing other criteria. Functionality is of paramount importance. A more efficient design is utterly worthless unless it is fully functional. It does not make sense to evolve on any other criteria until a fully functional circuit has been found. Therefore, it might make sense to evolve functionality before evolving with respect to other criteria. Functionality could come first in the evolutionary process, perhaps as the first stage of a multi-part evolutionary process.

The work of Thompson [30,31,32,33] first applied the idea of dividing the evolutionary process into stages in order to include multiple criteria. The criteria being considered for that work included fault-tolerance and, of course, functionality. The GA began by selecting circuits which were functional. After most of the circuits reached full functionality, the functionality tests for all circuits were expanded to include faults. The work of Coello [4] took a slightly different approach to the concept of stages. Instead of applying the stages to all circuits simultaneously, the stages were applied to each individual circuit. If a circuit was less than perfectly functional, then its fitness score was equal to its functionality. If a circuit was fully functionally, on the other hand, then its fitness score equaled its functionality score plus the score it achieved for the additional criterion. For instance, a circuit with sub-perfect functionality score of 7 and a size score of 5 would have a total fitness score of 7. A circuit with a perfect functionality score of 8, however, would have its size evaluation score—say it is 4—added to it for a total fitness score of 12. In this way,

circuits with perfect functionality would be selected before those without, even if they score worse on the additional criteria [4].

In his formulation of the problem, Coello [4] made use of the array method which was previously discussed (consult Section 4, figure 5). For most trials, the size of the array was five-by-five. The genotype encoding used an alphabet of cardinality five, with the relevant characters being the numbers 0 through 4. The genotype was composed of a series of three-character sub-strings, each of which represented a different position within the array. In each sub-string, the first two characters represented the positions in the previous column from which the represented position gets its inputs. The third character in the sub-string stood for the type of logic gate at that position. The sub-strings were concatenated to produce the entire genotype of a circuit. The concatenation went in from the top-leftmost position to the bottom-rightmost, first by row and then by column. The logic gates that were possible included AND, OR, XOR, NOT and a non-gate called WIRE, which simply carried through the logical value at its input.

5.1 Optimizing the Circuit-Size Evaluator

The inclusion of WIREs is important for the array formulation. It allows signals to be carried across the array without the need for extra gates. The inclusion of WIRE was also integral to the Coello [4] method of evaluating circuit sizes. Since each WIRE in the array represents a location without a logic gate, it was thought that counting the number of WIREs in the array would suffice for evaluating the size of the circuit. Moreover, counting WIREs in this way seems easier than analyzing the components to see which gates are actually being

used. Based on their success in optimizing logic circuits, there must be some truth to these assumptions. However, neither of these assumptions is completely true, and that gives way to an optimization of the Coello [4] algorithm which also constitutes a starting point for the research presented in this paper. The optimization presented here is a superior way to evaluate the size of a circuit. In turn, this leads to better, faster optimization of circuits by the GA.

The intuitive choice for determining the size of a circuit is to simply count the number of gates which are connected to the circuit. More precisely, one can count the number of gates in the array which are connected to the output, through any path. Any gate which is somehow connected to the output is part of the circuit, and probably has some effect on what the output is. It is a "used" gate. Any gate which is disconnected is completely unused in the determination of the output. Determining the size of a circuit by counting used gates is slightly more complicated than the wire-count method. For the wire-count method, the algorithm simply has to examine all positions in the array and count the number of wires present. For the used-gates counting method, it is necessary to examine the connections to see which gates are connected. This involves looking at the final gate to see where its inputs come from. The gates which feed into the final gate then have their inputs examined in the same fashion. Backtracking through the array like this, it is possible to find each gate that is somehow connected to the output. The used-gates method is harder to implement and may require more computation, but we will see that the benefits far outweigh these detriments. Specifically, the used-gates method allows optimizations to occur much more quickly.

The primary benefit of used-gates optimization lies in what happens with the unused gates. If the evaluation of circuit size is based only on the number of used gates, then the

extraneous gates in the array can become any gate without affecting the size evaluation. By contrast, the wire-count method pushes every non-essential gate toward changing into a WIRE. It is important that unused spaces in the array can take on different values without affecting the size evaluation, as with the used-gates method. If changes can be made to a circuit array without effecting its size evaluation, the circuit is said to lie on a neutral landscape; neutral landscapes have been shown to be very important for aiding biological and computational evolution [8,13,27,35,36].

The exact processes by which neutral landscapes aid evolution are not completely understood [27]. However, an intuitive example may clear up at least one way in which they can help. Consider a GA which is attempting to optimize a pool of potential circuits for size. As the algorithm proceeds, the mutation operator is constantly shifting connections within the circuits at random. This serves the purpose of randomly connecting new sections of the array to the used gates to see if there is any benefit. However, the wire-count method essentially ensures that there is nothing in other parts of the array besides WIREs. There is no extra logic in the array from which better solutions could be built. On the other hand, used-gates optimization allows for any gates to exist in other parts of the array. In most cases, a random mutation will not find anything useful in these extra gates, but occasionally it will. At least with used-gates optimization the chances are much higher for finding something useful in the extraneous gates, because they contain a variety of logic rather than mostly WIREs.

Used-gates optimization has been implemented before [35,36], but only in a pure-optimization context. It has never been combined with an algorithm which also searches for functional circuits. Nor has it been compared to the wire-count method directly. Later, we

will see how the used-gates method works as part of the complete GA. Here, the used-gates method was implemented to specifically compare its performance to the wire-count method.

5.2 Evidence for the New Size Evaluator

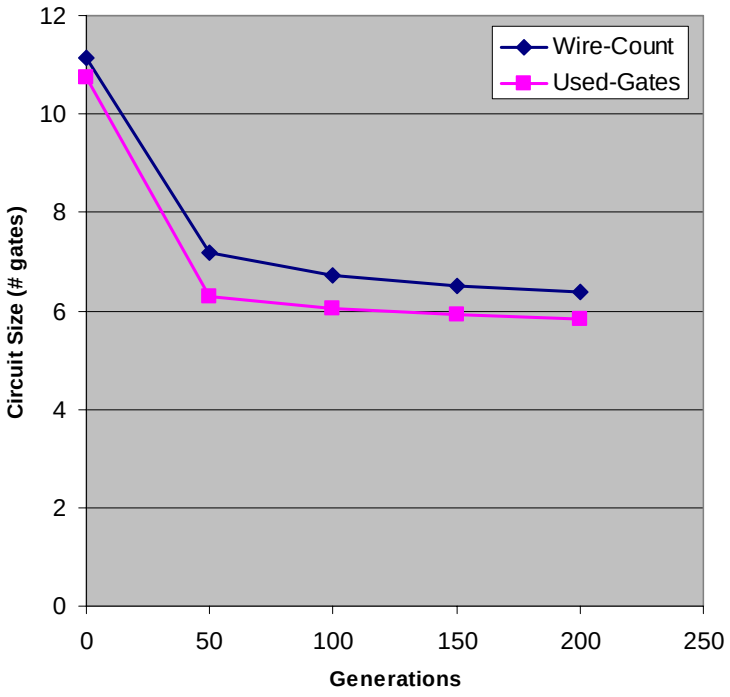

Figure 6. Performance of the size-optimization strategies.

Once used-gates optimization was implemented in the GA, trials were run to chart its performance against the GA using wire-count optimization. All trials were run on a Sun Microsystems Ultra 250, running SunOS 5.8. The algorithm was run using a crossover rate of 100% with a mutation rate of 0.4% and elitism. The generation size was set at 500 individuals. These values were set upon the basis of preliminary testing. It is not contended that they are optimal values, but it was determined that they work well. Three sets of 50 trials were run; each set used a different logic function as the goal. Within each set, 25 trials were run with wire-count optimization, and 25 were run with used-gates optimization. The three arbitrary functions were defined as follows, with respect to their minterms: $F_0 = \{6, 7, 10, 11, 12, 13\}$, $F_1 = \{5, 9, 10, 13, 14\}$, $F_2 = \{5, 6, 13\}$. The available logic gates were AND, OR, NOT and XOR. The array formulation was used to represent the

functions, with each array having 5 rows and 5 columns. Using the systematic search method, described later in Section 6, it was possible to determine that the smallest possible circuit for each function under these conditions used 4 gates. Measurements corresponding to the smallest circuit produced by the GA were gathered for each trial at 50 generation intervals. The first interval began after the first fully functional circuit appeared, since size measurements are irrelevant up to that point. These circuit size measurements were averaged across all used-gates trials and all wire-count trials, so that a comparison could be made between the different optimization techniques. The results are shown in figure 6.

To determine the effect of extra computation required by the used-gates technique, the GA was run 5 more times for each optimization technique. The GA was allowed to run through 200 generations on 5 different functions. A measurement of the actual time associated with each run was recorded using the Unix shell "time" command. The times recorded showed only small variation, but an average was taken across the trials for each optimization technique. The average time for wire-count to run was 28.46 seconds, while the time for used-gates was 29.60 seconds. This constitutes a 3.85% increase in time and corresponds to wire-count working through 3514 potential

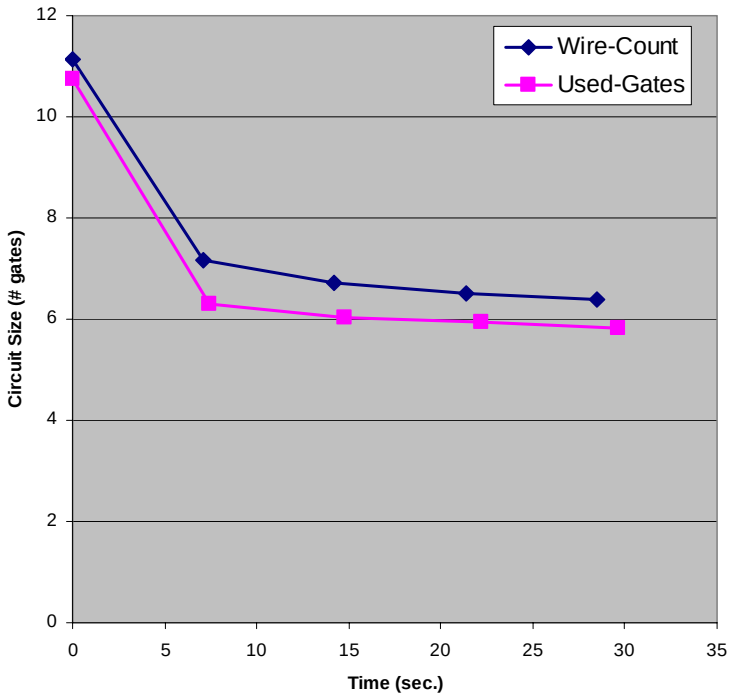

Figure 7. Normalized size-optimization performance.

circuits per second, while used-gates works through 3378 circuits per second. This result was then used to normalize the findings in figure 6, by putting time across the x-axis, rather than generations. This is an important normalization to perform, since the time to work through a generation apparently takes longer for used-gates optimization. The normalized graph is shown in figure 7.

It is clear from figure 7 that the used-gates method is superior to the wire-count method. On average, it allows smaller circuits to be achieved more quickly. The effect is seen immediately as the optimization process begins, and is maintained throughout. Moreover, this holds true despite a 3.85% increase in time. The two lines in figure 7 appear to show a slow convergence. It is assumed that they would converge, eventually, at 4 since it is the minimum circuit size. Nevertheless, the used-gates techniques offers a clear advantage in getting down to the minimum circuit size, or to any circuit size.

5.3 Optimizing the Functionality Evaluator

The second optimization being presented here deals with evaluating the functionality of a circuit. Evaluating functionality is always a matter of presenting a circuit with each of the possible inputs, and examining the outputs that result. For instance, if there is a 3-input circuit which needs to be evaluated, logic dictates that there are 8 possible combinations of inputs values. Thus, the evaluation would involve presenting each of the 8 combinations, one after the next, and seeing what the outputs are. The 8 actual outputs are then compared to the 8 expected outputs. Usually, the functionality of the circuit is considered to be the number that match up. If our 3-input example circuit produces 7 of the 8 expected outputs

correctly, then its functionality evaluates to 7. If it only gets 1 right, its functionality would be evaluated to 1. This matching method is easy and simple, but perhaps also slightly naïve.

With the naïve method for evaluating functionality, a circuit which produces the wrong outputs for every input combination is ranked as the worst circuit. This essentially implies that that circuit would need to undergo the most changes to become completely functional. It is important to realize that this assumption is wrong. Most formulations of logic circuit design include the possibility for using a NOT gate, although some notable exceptions occur (i.e. using all NANDS or all NORS). If NOT can be used in the design, then the "worst" circuit described above is actually not worst at all. In fact, if a NOT gate is attached to its output, then the so-called "worst" circuit becomes fully functional. Rather than being the worst circuit, it is very near to being perfect. The similar situation exists with any circuit which produces only 1 correct output. It is presumably close to the circuit which produces 0 correct outputs, which is very nearly a perfect circuit. If this reasoning is continued, it seems like the worst circuits are actually those which produce half correct outputs and half incorrect.

Based on this reasoning, a new functionality evaluator was developed. As an example, supposed we want a GA to design a circuit which produces some 3-input function. The old evaluator would have ranked a circuit producing all outputs perfectly as an 8, and a circuit producing no correct outputs as a 0. Thus, circuits would fall onto the scale which, in terms of how many outputs matched, looked like this: 0,1,2,3,4,5,6,7,8. With the new evaluator, in terms of how many outputs matched, looked more like this: 7,6,5,4,4,5,6,7,8. That is, the worst ranked circuits were the ones which produced about half of the outputs correctly. This idea was implemented in a GA and tested.

5.4 Evidence for the New Functionality Evaluator

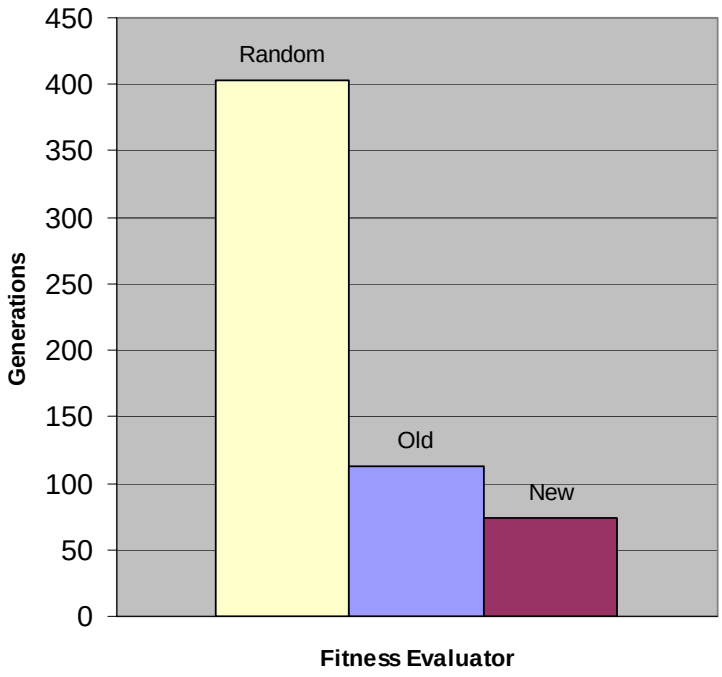

Figure 8. Performance of the fitness evaluation strategies.

Once the new functionality evaluator was implemented in the GA, trials were run to chart its performance against the old one. As before, trials were run on a Sun Microsystems Ultra 250, running SunOS 5.8. As before, the algorithm was run using a crossover rate of 100% with a mutation rate of 0.4% and elitism. The generation size was set at 500 individuals. Three sets of 50 trials were run; each set used a different logic function as the goal. Within each set, 25 trials were run with the new evaluator and 25 were run with the old. The three arbitrary functions were defined as follows, with respect to their minterms: $F_0 = \{6, 7, 10, 11, 12, 13\}$, $F_1 = \{8, 10, 13\}$, $F_2 = \{5, 9, 10, 11, 12, 13\}$. The available logic gates were AND, OR, NOT and XOR. As before, a 5-by-5 array formulation was used for the circuits. Measurements were taken of the number of generations it took for the GA to obtain one fully functional circuit using each evaluator. These measurements were averaged across trials and compared between both evaluators. Measurements were also taken for a third evaluator which acts a benchmark.

This third evaluator randomly assigned a functionality score to each circuit, rather than following any systematic evaluation procedure. The results are shown in figure 8. As compared to the old evaluator, the new one allowed the GA to achieve a fully functional circuit in 34.5% fewer generations.

To determine the effect of extra computation required by the new evaluator, the GA was run 5 more times for each optimization technique. The GA was allowed to run through 200 generations on 5 different functions. A measurement of the actual time associated with each run was recorded using the Unix shell "time" command. The times recorded showed only small variation, but an average was taken across the trials for each optimization technique. The average time for the old evaluator to run was 26.24 seconds, while the time for the new one was 26.34 seconds. This constitutes a 0.38% increase in time and corresponds to the old evaluator working through 3811 potential circuits per second, while the new one works through 3797 circuits per second. This result was then used to normalize the findings in figure 8, by putting time along the y-axis, rather than generations. This normalization is important, since the time to work through a generation takes slightly longer for the new functionality evaluator.

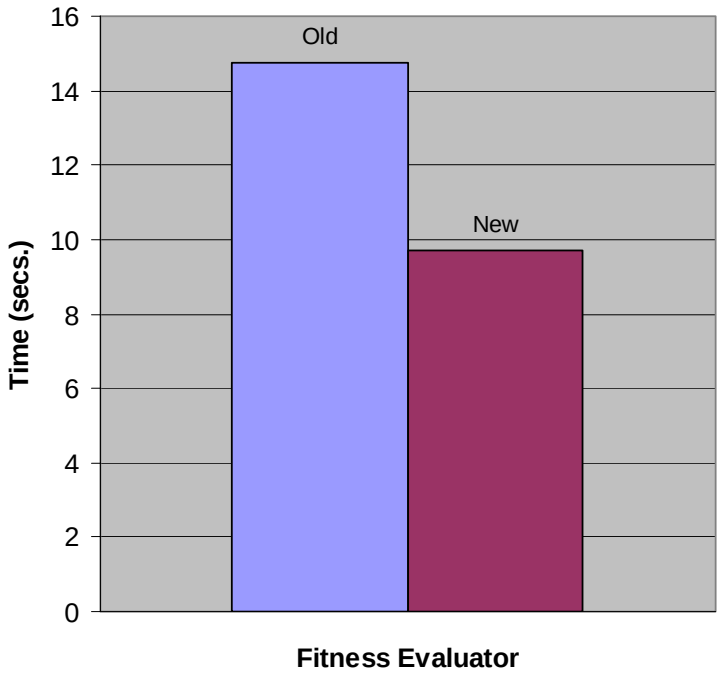

Figure 9. Normalized fitness evaluator performance.

The normalized graph is shown in figure 9. Even with the normalization, the new evaluator still allowed the GA to achieve a functional circuit in 34.2% fewer generations.

Both the old functionality evaluator and the new one offer a clear advantage over the randomized version. For both the new and old evaluators, the improvement is well over 75%. From figure 9, one can also clearly see that the new functionality evaluator is superior to the old one. It allows functional circuits to be designed much more quickly, on average. The improvement is dramatic enough that it far outweighs the 0.38% increase in time required for additional computation. The new functionality evaluator distinguished itself as the clear choice for speeding up the operation of the GA.

5.5 Combining the New Optimizations

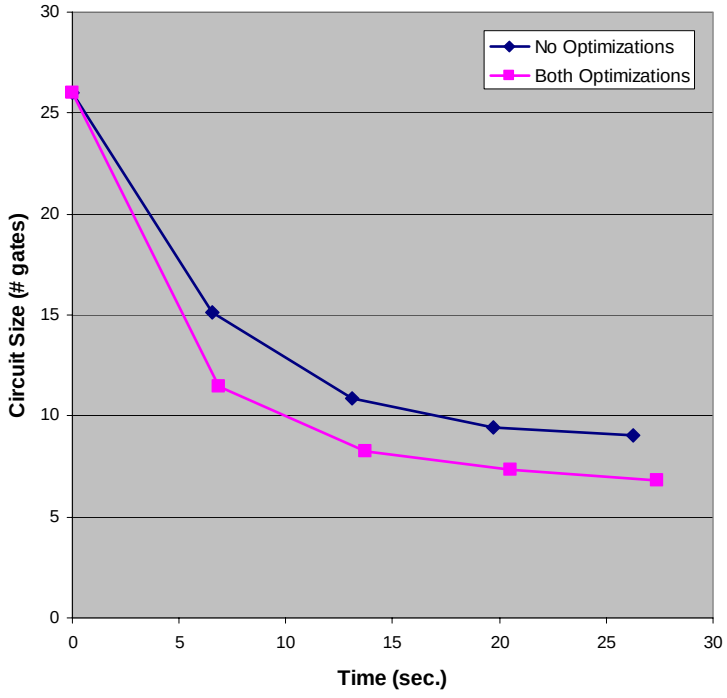

Figure 10. Normalized GA performance with both optimizations.

Finally, both of the optimizations were put together into one algorithm. The purpose of this was to see if the two optimizations were compatible with each other. That is, it was important to make sure that the GA was still improved once both of the optimizations were incorporated into the

algorithm. It was considered unlikely that the optimizations would conflict with each other, since they occur at separate stages of the evolutionary process. However, it gives peace of mind to see the two optimizations working side-by-side.

Trials were run on a GA which implemented the new functionality evaluator and the used-gates optimization technique. All search parameters were kept identical to the trials previously described. Two sets of 50 trials were run; both sets used a different logic function as the goal. In each set, 25 trials were run with neither of the new implementations and 25 were run with both. The functions used were as follows: F_0 = {6, 7, 10, 11, 12, 13}, F_1 = {5, 9, 10, 13, 14}. Measurements corresponding to the smallest circuit produced by the GA were gathered for each trial at 50 generation intervals. These circuit size measurements were averaged across the optimized and non-optimized conditions. The averaged data were then normalized to account for the increased time needed to compute the newly implemented techniques. The normalized results of the trials can be seen in figure 10. The results are as expected. When the new functionality evaluator and the used-gates optimization techniques are implemented together in one GA, the algorithm operates substantially more efficiently.

6 Developing Systematic Search

The second part of research in this paper uses systematic search to automate logic circuit design. The specific search method employed is a version of BFS. Earlier, we discussed the advantages and disadvantages associated with both types of search: systematic search and local search algorithms like GAs. The major advantage of systematic search methods like BFS is that they can return optimal solutions. Indeed, with careful problem

formulation, one can ensure that BFS will return optimal solutions every single time. Fortunately, it is not difficult to formulate most problems so that BFS can meets these guarantees.

6.1 Skepticism About Systematic Search

Unfortunately, however, the benefits of BFS come at a high cost. The space and time requirements can be prohibitively large, making this method of searching totally impractical for some problem instances. Problems like logic circuit design involve a huge number of combinations to consider. Not only do systematic search strategies like BFS need to consider all possible combinations, but they also need to keep their entire search history in memory until a goal is found. It is easy to imagine that the high resource requirements could easily overwhelm any attempts to employ systematic search. This may be the reason why systematic search has never been tried for this purpose; it seems highly impractical.

Another potential reason why systematic search has never been used to automate logic circuit design is that the problem formulation is unusual. Conventionally, the problem is essentially formulated as a rapid examination of whole potential circuits. The potential circuits are examined one at a time until the desired one is discovered. In this formulation, all that matters is the final circuit. The path to that final circuit simply represents the design process of configuring a desirable circuit. Once the desirable circuit is found, the path is irrelevant. Systematic search does not fit with this formulation. The path to the goal always matters in systematic search. In order for the path to the final circuit to be relevant, a new formulation was developed for the problem of logic circuit design.

6.2 Re-Formulating Logic Circuit Design

The new formulation hinges on one important idea: the path to the goal must represent the desired circuit. If the path to the goal represents a finished circuit, then it may be unclear what the search nodes represent. It may not be immediately obvious why, but in this research the search nodes were conceptualized as truth-table columns. Initially, there are several nodes present in the search tree. Each initial node represents the truth-table column for one of the input variables (figure 11). Just as a conventional truth table is seeded with columns that represent every possible combination of the inputs, so is the search space initially seeded with search nodes representing the truth-table columns of the inputs. To forward the search, nodes are not expanded in the conventional sense in this formulation. Rather, new nodes are formed by selecting two existing nodes and a logic operation. This is similar to the way new columns are formed in a conventional truth table. After selecting, the logic operation is then applied

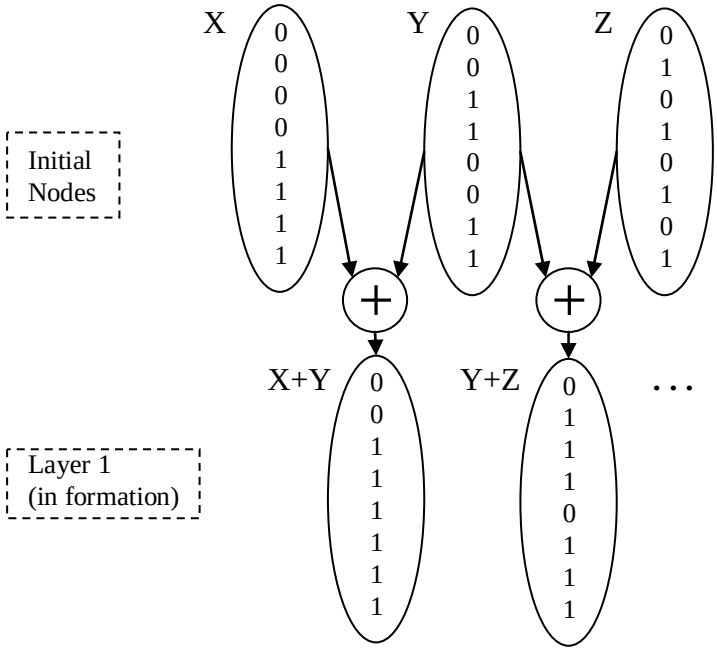

Figure 11. Systematic search as applied to logic circuit design. The initial nodes are the input variables. Nodes are combined, via logical operators, to create new nodes.

to the corresponding rows in the chosen nodes, and a new truth-table column is formed which constitutes a new search node.

The goal is reached when the newly formed node has a column which matches the truth table of the desired function. Note that the goal node does not represent the desired circuit, but rather the output of that circuit. This is exactly the same as the final column of a conventional truth table, which represents the output behavior of a function, rather than the function itself. When the goal node is discovered, the exact path which the search used to get to that goal is remembered. The particulars of the path can be constructed by traversing backwards up the tree and examining the logic operations that were used in arriving at the correct output. The path contains all the important information about the types of operations used and the order in which they were applied. In other words, the path represents a circuit which can produce the desired output.

This kind of search can find circuits of optimal size. Indeed, it can be adapted to find optimal circuits for many different criteria. This is done by maintaining the layer-by-layer approach that BFS follows. Within each layer, or phase, the only nodes that are allowed to enter the search tree are those of a given quality. The following phase allows nodes of a slightly lower quality to be examined, and so on. The simplest example to consider is searching for circuits of the smallest depth. Consider the initial nodes, which represent the circuit inputs. These initial nodes require no logic gates to create and, therefore, have a depth of zero. The first search phase would involve the creation of all possible nodes that can result from combining these initial nodes. In other words, this first phase examines all possible combinations which have a depth of one. Phase two of the search would again involve all combinations of existing nodes. This time, that means all combinations of the

initial nodes and the nodes produced during phase one. A given phase-two combination may put together an initial node with a phase-one node, or it may put together two phase-one nodes. Either way, the resulting nodes will have a depth of two. Similarly, the nodes produced at phase three will have a depth of three, and so on. The search continues until some node is created which has the desired truth-table column.

For the previous example, it is important to realize that the results of a particular search phase do not simply constitute *some* nodes of a particular depth, but *all* possible nodes of a particular depth. By examining all possible nodes of a given depth before moving on to the next phase, it is ensured that the first goal node reached will have the shortest possible depth. No node with a shorter depth is possible, and we are sure of this because every single node of shorter depth was already created and examined.

It is also important to make a note about redundant search nodes. A redundant node is one which contains the same truth-table column as some already-formed node. With this systematic search strategy, it is important to eliminate redundant nodes which arise. More accurately, each new node should be checked for its redundancy status before being added to the search tree. The redundant nodes will make the search proceed more slowly. In terms of finding a goal node, it is sufficient to have only one node representing a given truth-table column. Any additional copies of that column will simply cause unnecessary combinations to be considered. Even worse, it is possible for redundancies to lead the search towards a sub-optimal answer. This would result if the redundant node occurred at some phase later than the original node. If this occurred, then any solution involving the redundant node would have a superior version, involving the original node. Thus, it is essential eliminate redundant nodes and avoid this potential snag.

Adapting this method to search for other criteria, besides depth, is not very difficult. We have seen that circuit size is an important criterion to consider, and searching for optimal size does not require drastic change to the algorithm. In fact, the search can proceed almost identically to the way it did for the depth-concerned example. At each phase, all possible combinations of nodes will still be considered. The difference will be in which nodes, produced during a given phase, are allowed to join the tree. For the depth-concern search, a phase allowed all non-redundant nodes of a given depth to join the tree. The reason the depth-concerned search is so simple is because all nodes created during a given phase are already of the specified depth. This is not the case for a size-concerned version of the algorithm. If circuit size is being considered, a phase should allow only nodes representing circuits of a given size to join the tree. Ensuring that this happens is slightly more complicated. For example, the third phase should presumably allow all nodes representing circuits of size three onto the tree. However, phase two will have allowed nodes of size two onto the tree. If two size-two nodes are combined, then a circuit which is too big will be produced. To solve this, each node must have some additional information kept with it about how many gates its path has involved. This is a simple matter of adding together the number of gates that each parent node used, plus one for the gate combining the parents. With this information, nodes can be checked to see what their size is before they are allowed to join the tree. Any node representing a circuit that is too big for the current phase must be temporarily discarded and kept from the tree. If the search continues, it may be regenerated and allowed to join the tree later in the search.

6.3 Implementing Systematic Search

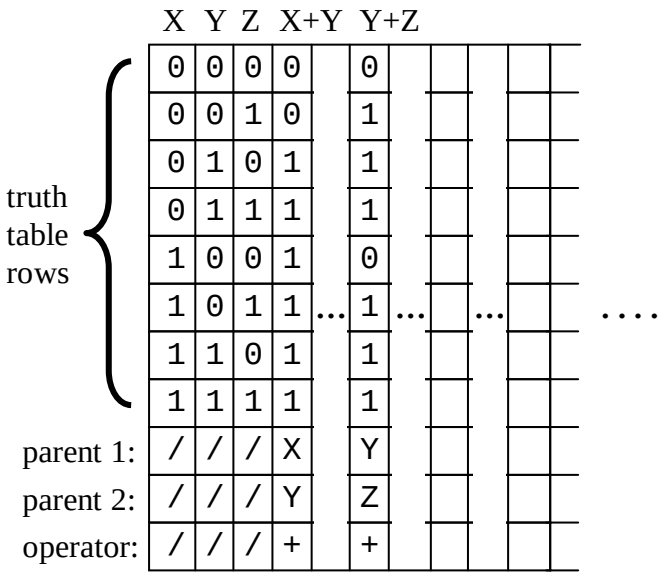

Figure 12. Systematic search as it was implemented for this paper, using a 2-D array to store the nodes.

This systematic search strategy can be implemented quite easily using a two-dimensional array (figure 12). Each array column can hold a truth-table column from a single node. Extra rows can be made below each column to hold the additional information about parents, depth/size, etc. The first columns of the array are initially seeded with the truth-table columns of the inputs. Creating new nodes is simply a matter of selecting two columns and a logical operator, and applying the operator to each row of the columns. When new nodes are created, their respective truth-table columns can be placed in the next open column of the array.

For the purposes of this paper, both a depth-concerned version and a size-concerned version of systematic search were implemented in Java. The goal state was implemented, simply, as another array containing the desired truth-table column. A hash table was also maintained which tracked those columns that had already been generated. In this way, checking for redundancies could be done in constant time even as the number of columns

grew. Implementation and later trials were run on a Sun Microsystems Ultra 250, running SunOS 5.8.

Once the depth-concerned and size-concerned versions of systematic search were implemented, they were used to generate circuits for several arbitrary, 4-input logical functions. For some functions, such as the one featured in figure 13, there is a clear difference between the smallest-depth circuit and the smallest-size circuit. The circuit produced by the size-concerned version of the algorithm produced a circuit which uses only seven gates to accomplish the correct output. The depth-concerned version produced a circuit of size nine, which is still fairly small but not optimal in size. The larger circuit is, however, optimal with respect to depth, having a depth of merely four. This, as opposed to the smaller circuit which has a depth of six. There are instances, however, when the two criteria appear not to be mutually exclusive. For example, figure 14 shows a pair of functions implementing the same logical function. The size-concerned one uses only seven gates, but its depth is five. The circuit produced by the depth-concerned search also uses seven gates, and its depth is only four.

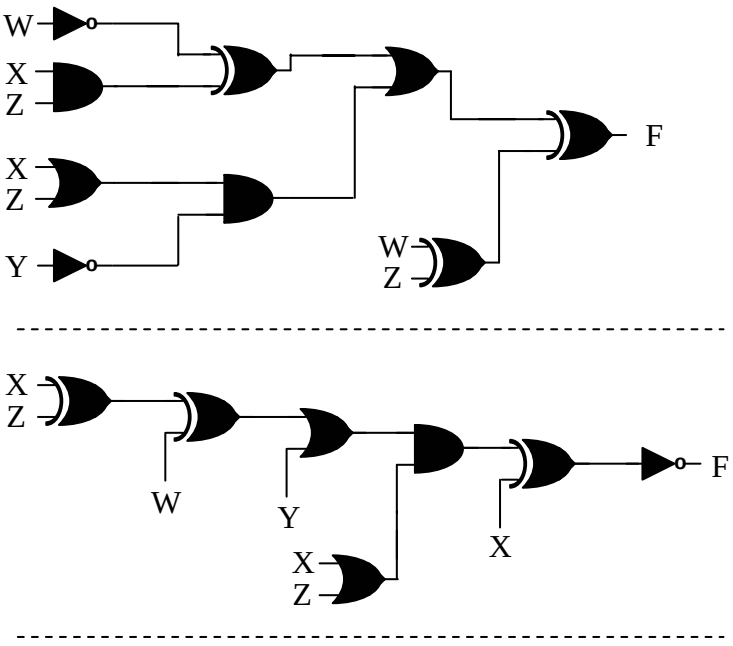

Figure 13. Circuits produced by the depth-concerned (above) and size-concerned (below) versions of systematic search. The function is F(W,X,Y,Z)={0,2,4,6,7,8,9,10,13,14,15}.

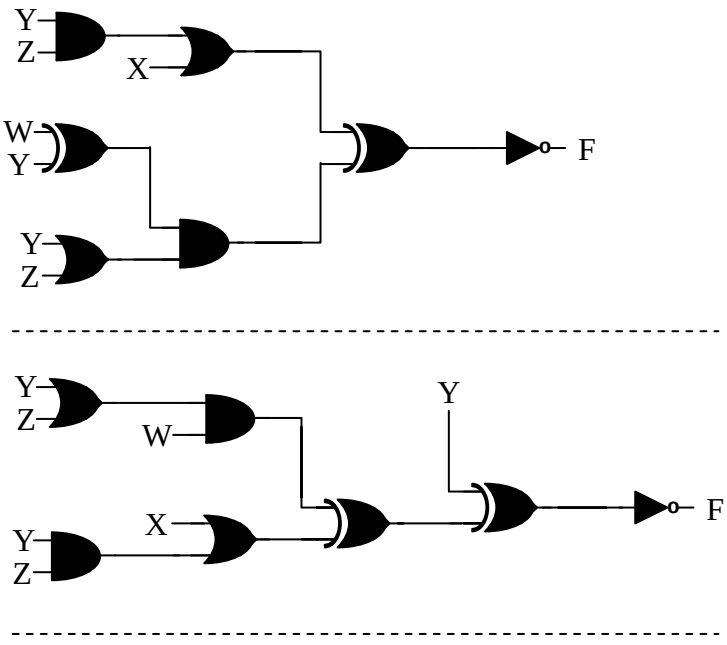

Figure 14. Circuits produced by the depth-concerned (above) and size-concerned (below) versions of systematic search. The function is F(W,X,Y,Z)={0,1,3,6,7,8,10,13}.

The circuits presented in figures 13 and 14 are, as specified, either the smallest or the shallowest circuits possible for the specified functions. That is not to say that they are unique. It is possible that other circuits of equal size and/or depth could be produced. However, no circuit implementing these functions can be smaller than the size-concerned circuits presented and none can be shallower than the depth-concerned circuits presented. Systematic search of this kind will always return the optimal solution, which is the major advantage of this type of search. Moreover, it should be noted that the solutions provided by systematic search can be substantially superior to human designs. Previous work [4] which used the example functions presented here reported that the smallest human-designed circuit for the function in figure 13 involved eleven gates. Likewise, the smallest human-designed circuit for the function in figure 14 used nine gates. In both cases, the optimal function discovered by systematic search was substantially smaller (36.4% and 12.5% smaller, respectively), with sizes of seven gates.

One surprise that was observed after the algorithm was implemented was that its time requirements for completing were not as high as expected. All solutions for the examples above were obtained after relatively short amounts of time. The depth-concerned circuit in

figure 13 was obtained after 12 seconds of runtime. Even better, the depth-concerned circuit in figure 14 was gotten after only 3 seconds. As for the size-concerned circuits, both took 23 seconds of runtime for the algorithm to find them. These numbers did not seem outrageously large, as it was assumed they would be. In fact, they did not seem large at all. After this observation, it was thought that the systematic paradigm might not be so impractical for these purposes. However, similar experiments on 3-input functions suggested that this optimism might be unfounded. In these casual experiments, solutions to a wide range of 3-input functions were found in considerably less than one second. Although this kind of speed is a good thing, the disparity between times for the 3-input and 4-input cases suggested that scaling might be a serious concern with this algorithm. If searching for 4-input functions takes an order of magnitude longer than searching for 3-input functions, the times for the 5-input case might begin to illustrate the expected problems with systematic search. In other words, even though systematic search looked very promising at finding small circuits, problems might begin to arise when trying to scale up to more complex circuits. This issue is explored in Section 7, along with an exploration of this issue for GAs. This is done as a comparison of the two search techniques, in an attempt to determine which one is superior for logic circuit design.

7 Comparison of Relevant Search Techniques

As we have seen, automating logic circuit design is desirable. Ideas from artificial intelligence have been successfully applied to this problem, and automation has become a reality. Both genetic algorithms and systematic search are highly capable of designing logic

circuits with good results. However, it is still relatively unclear which algorithm is best for this purpose. It may be that neither algorithm is entirely better; there is probably a trade-off associated with selecting one algorithm over another. As such, many factors need to be considered so that maximum benefits can be gained when the trade-off is made. The relevant aspects of the trade-off are time requirements, memory usage, solution quality and scaling.

7.1 Solution Quality

To some extent, we have already seen how solution quality differs for GAs and systematic search. Systematic search is guaranteed to give optimal solutions, which has two major advantages. First, optimal solutions are obviously the most desirable solutions. However, that fact alone does not give systematic search an advantage over GAs. Genetic algorithms can also return optimal solutions in many instances. Indeed, they are fully capable of doing so. It is the optimality guarantee that systematic search provides which is truly an advantage. By searching systematically, it is guaranteed to find an optimal solution. GAs, on the other hand, deal only with localized portions of the search space at any given instant, and therefore cannot provide any guarantees. Although the solution returned by a GA might be optimal, there is no way for the GA to recognize and confirm that fact. One must simply trust that GA has performed well enough, or verify the solution through some other means.

There is a related consideration which is more of a practical matter. A GA working on logic design has no idea when to halt, because the proportions of an optimal solution are not known ahead of time. Therefore, the GA doesn't know when its solution is good enough

that it can return it. Rather, it must be cut off after a given amount of time and forced to return the best circuit developed up to that point. This is both good and bad. On one hand, it is possible to stop the algorithm at any point during the optimization process and get an intermediate solution. This, as opposed to systematic search, which must run to completion before it has any kind of solution at all. On the other hand, there must be a judgment call about when the GA has progressed far enough to provide a satisfactory solution. Without a pre-conceived notion of what an optimal solution looks like, it is unclear what the basis for cutting off the algorithm would be. It can be based on inspection by a person and a "good enough" determination being made. It might also be based on a pre-determined, *ad hoc* time allotment. Either way, the algorithm might be cut short of finding a superior solution, or alternatively, it might be allowed to run for extra time after an optimal solution is found. Systematic search will not have these troubles; it will run just as long as it must.

7.2 Time Requirements, Considered Empirically

Some equally important considerations are those of time requirements and memory usage. For smaller circuits, which most other research to date has considered, neither of these has caused serious concern. Both time and space requirements have been reasonable. However, it is critical to consider these with respect to scaling the problem up. Designing more complex circuits with more inputs should be the eventual goal of automated circuit design. Large, complex circuits are the most challenging to design, but potentially the most useful also. Some theoretical work has suggested that finding optimal logic circuit designs is NP-Hard [20]. Therefore, scaling poses a serious problem for automation, in that the space

and time requirements might grow exponentially with increasing inputs, thereby becoming quite large. In practice, the few attempts to use GAs for increasingly complex circuits have been problematic. This is evident in some work that has applied GAs to circuit design [4,35,36].

Inuitively, it is unclear whether GAs or systematic search should be better in terms of time requirements. Thus, a series of trials were run to test the time requirements of both algorithms across circuits with increasing numbers of inputs. The input sizes considered ranged from 1 variable to 4 variables. The GA used for these trials incorporated both of the optimizations described in Section 5. In all cases, the desired functionality of the circuit was to have a 0 for all minterms except the final one. For instance, the 3-input function was defined as F = {8}, while the 4-input function was defined as F = {16}. The functions were meant to be simple, while still illustrating the increasing complexity associated with increasing inputs. Selecting the final minterm ensures that each input will be relevant to specifying the behavior of the function. The minimum number of gates that can be used to implement each of these functions is equal to the number of inputs minus one. For instance, the 4-input function, F = {16}, can be

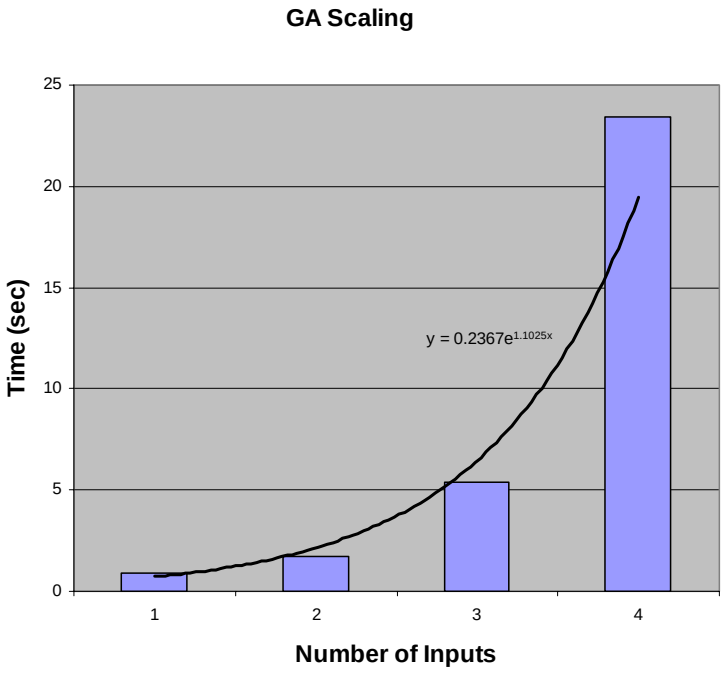

Figure 15.

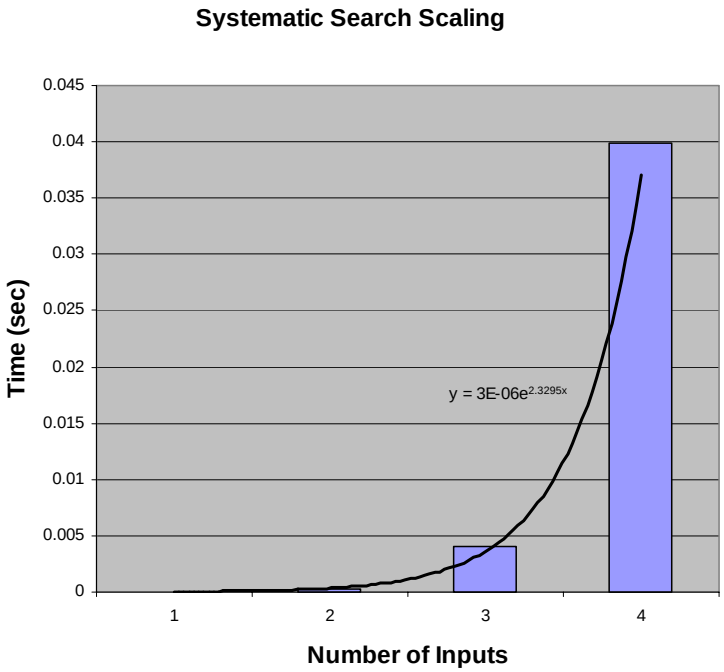

Figure 16.

implemented with no less than 3 gates. In all trials, including those with the GA, the trial was stopped when a circuit with the minimum number of gates was achieved.

The results of the trials can be seen in figures 15 and 16. Both graphs show exponential increases in the time requirements as the number of inputs increases. This fits in with the work suggesting that logic circuit design is NP-Hard [20]. Trend lines added, using Microsoft Excel™, suggest that the time requirements of systematic search grow faster than those of the GA (note their exponents of 2.33 and 1.10, respectively). Note, however, that if the number of inputs stays small, the time requirements for systematic search are quite a bit lower than those for the GA. Extrapolating from the trend lines would imply that a number of inputs less than 10 would show this same kind of behavior. However, beyond that point, the trend lines cross and systematic search becomes the slower algorithm.

7.3 Space Requirements

Not much concern has surrounded the memory usage associated with automating circuit design, and it is easy to understand why. GAs have been the algorithm of choice, and within one run of the algorithm, the space requirements are constant. When each new generation is created, the original generation is simply erased from memory. Thus, the memory requirements never grow as the algorithm proceeds. In terms of scaling, there is not necessarily any reason to increase the number of individuals per generation. There is, however, a possibility that the genotype strings will need to be lengthened to accommodate more logic gates. Each additional gate only adds a tiny amount to each string, however. For instance, a string representing 25 gates would be 75 characters long, using our formulation of 3 characters per gate. The first additional gate, then, would add only 3 more characters to the string of 75. This kind of memory increase is dwarfed by the memory requirements of systematic search. Even if we assume that the number of gates will double with every additional input, which should be very generous, GAs still win out in terms of memory requirements as the scale increases. The reason for this is that systematic search, as it is formulated here, must have enough space in memory to store every possible truth table column for a given input size. Despite the fact that the algorithm removes redundant columns, this number still grows astoundingly fast as the number of inputs increases. To be exact, the number of possible columns for a truth table of n inputs is 2 raised to the power 2^n. For the 3-input case, this number is only 256. The 4-input case requires enough space to store 65536 columns, and the 5-input case already requires space for over $4*10^9$ columns.

The requirements quickly become so prohibitively large that GAs clearly win out in this respect.

7.4 Trade-Off Conclusions

The space requirements of systematic search are so large that GAs appear to win out altogether. Such growth would seem to keep the algorithm from being implemented at all, and therefore trump all other considerations. However, it is clear that this is not the case in all situations. For circuits with less than five inputs (and, potentially up to five inputs), systematic search is the superior choice. Not only are its space requirements comparable to those of GAs, but the amount of time it takes to complete is much lower, and it is guaranteed to give the optimal solution. The superior time requirements for systematic search would allow it to be the better choice for up to the 10-input case, but the space requirements grow quickly enough as to make other considerations irrelevant. For designing these larger circuits, GAs appear to be the better choice.

8 Conclusions and Future Work

There is much promise in applying the search techniques of artificial intelligence to the problem of automating logic circuit design. Conventional design techniques, such as those to aid human designers, can lead to good designs, as well. However, these conventional techniques can often constrain the space of possible designs too tightly. Often, these methods use Boolean standard forms as their templates, which does not allow for a full

range of possible configurations. By broadening the range of possible designs, it is possible to find even better solutions. Indeed, it is possible to find circuits which are the smallest, shallowest, or otherwise most optimal circuits for a specified logical function. It is also possible to produce extremely novel circuit designs. The downside of allowing a wider range of circuits is that it quickly makes the design problem impractical for human designers to participate. There are simply too many possible circuit configurations to consider. The good news, though, is that computers are extremely proficient at considering large numbers of configurations efficiently. In fact, that is essentially the purpose of AI search techniques. Genetic algorithms have been the common choice among search techniques for this purpose. Using GAs, it has been possible design logic circuits which are superior to those produced conventionally and to do so efficiently. This paper examined specific ways to make GAs more efficient for this purpose. Also, it examined the application of systematic search to circuit design, a type of search which had previously not been used for this purpose.

8.1 Conclusions

As seen in Section 5, the work of Coello [4] successfully evolved logic circuits by splitting a GA into two phases. One phase was for evolving functional circuits, while the second pushed towards smaller circuits. The research in this paper presented optimizations for both of these phases. The functionality-phase optimization relies on overcoming the naïve concept of functionality evaluation. This old method simply counts how many errors a circuit produces. By contrast, the new method takes into account how adding new gates might change the circuit's output. In other words, the new evaluation more accurately

portrays the amount of change a semi-functional circuit must undergo before it is fully functional. Although this new functionality evaluation does take more time to compute, the time gains far exceed the costs. This overall improvement was confirmed empirically, across many trials with multiple logical functions.

The second, size-oriented phase of the GA was improved as well. In order to find smaller circuits, one must accurately assess the size of the circuits being considered. The optimization presented here more accurately determines the size of circuits by counting the number of gates that a circuit utilizes. Intuitive as it might sound to simply count the number of used gates, this approach had not previously been implemented in a GA of this type. The reason is mainly that it involves more intricate analysis of the circuit to determine which gates actually contribute to the output. Older methods took a less computationally-intensive approach, which was also less accurate. The method presented here does require more computation, but this is outshone by the efficiency with which better circuits were produced. Again, this improvement was illustrated empirically.

A formulation was also presented for applying systematic search techniques to the design process. To the extent that no research on this topic could be located, it is contended that this had never previously been attempted. It is believed that this is, in part, due to the unusual problem formulation required to support systematic search. Specifically, the search path itself represents a circuit. The major benefit of using systematic search is a guarantee that optimal solutions will be found. Indeed, this technique can be used to find the best known circuits for any specified functions. The search criteria can be changed, as well, to accommodate different criteria for optimality. The criteria explored here were circuit size and depth.

The downside of systematic search should be that its resource requirements (i.e. time and memory) are higher than with local search algorithms like GAs. At least, that is generally the case when search algorithms are used for other problems. However, after observing the behavior of both algorithm types, it was found that this was not always the case for the logic circuit design problem. Systematic search actually requires less time than GAs when searching for less complex circuits (i.e. fewer than 10 inputs). Systematic search also requires comparable memory to GAs for less complex circuits (i.e. fewer than 4 inputs). Taking into consideration the benefits of systematic search, it seems that these methods are preferable for designing less complex circuits.

As a matter of scaling up to more complex circuits, though, systematic search becomes impractical. Time requirements scale poorly, but it is memory requirements that are of even greater concern. Therefore, it appears that GAs are the better choice when searching for circuits having more than 5—or possibly 4—inputs. It is crucial to note that GAs are not necessarily ideal, though. Their time requirements scale exponentially with increasingly complex circuits, which is less than desirable. Thus, scaling is a problem for both types of algorithms, even though it is less of a problem for GAs.

8.2 Future Work

In the future, research on automating logic circuit design must focus on the problem of scaling. If the scaling issues are not overcome to some extent, these algorithms will remain largely impractical for designing more complex and interesting circuits. How to overcome this challenge is a very open question. To some extent, it might be possible to

overcome the scaling issues with systematic search by the design of a good heuristic. Good heuristics could allow systematic search to proceed more directly to the goal, using less time and memory along the way. Designing heuristics for this purpose might prove rather difficult, however, since measures for the distance between sub-functional circuits and completely functional circuits are rare. However, the optimized functionality evaluator presented in Section 5 might be a starting point for such a heuristic. It attempts to get at such a distance measure.

The optimized functionality evaluator presented in this paper also lends itself to further refinement. It is, perhaps, only the first step in breaking away from the more naïve approach. In evaluating the functionality of circuits, it accounts for the ability of the NOT gate to complement a circuit's output. Future evaluators might equally account for other gates' abilities to change a circuit's output. Other evaluators might even account for some combination of gates, and find that to be even more useful. Such evaluators should be implemented and tested empirically to see whether they improve GAs.

Finally, it may be possible to use the resulting circuits from these search algorithms to infer new design principles that human designers can use. Patterns in the design process and in the results of automated design might be emulated to improve the quality and efficiency of hand-designing methods. Finding those patterns is beyond the scope of this paper, but is certainly worthy of examination. Until the scaling and other issues can be resolved for automated circuit design, human beings will still need to do the hard work of designing by hand. If any new principles for designing can be added to their repertoire by examining the results of automated search techniques, it would undoubtedly be a useful result of this area of research.

REFERENCES

1. Bennett, F.H. III, Koza, J.R., Andre, D. and Keane, M.A. (1996). "Evolution of a 60 Decibel Op Amp Using Genetic Programming." 1st Intl. Conf. on Evolvable Systems, 455-469.

2. Boole, G. (1854). An Investigation of the Laws of Thought. MacMillan & Co.: Cambridge, England.

3. Canham, R.O. and Tyrell, A.M. (2002). "Evolved Fault Tolerance in Evolvable Hardware." Evolutionary Computation 2002, 1267-1271.

4. Coello Coello, C.A., Christiansen, A.D. and Aguirre, A.H. (2000). "Towards Automated Evolutionary Design of Combinational Circuits." Computers & Electrical Engineering 27, 1-28.

5. Dwyer, R. (2003). Genomic Perl. Cambridge University Press: New York.

6. Gajski, D.D. (1997). Principles of Digital Design. Prentice Hall: New Jersey.

7. Garvie, M. and Thompson, A. (2003). "Evolution of Self-Diagnosing Hardware." 5th Intl. Conf. on Evolvable Systems, 238.

8. Harvey, I. and Thompson, A. (1996). "Through the Labyrinth Evolution Finds a Way: A Silicon Ridge." 1st Intl. Conf. on Evolvable Systems, 406-422.

9. Holland, J. (1975). "Adaptation in Natural and Artificial Systems." Univ. of Michigan Press: Ann Arbor.

10. Horn, J., Nafpliotis, N. and Goldberg, D.E. (1994). "A Niched Pareto Genetic Algorithm for Multiobjective Optimization." Evolutionary Computation 1994, 82-87.

11. Iba, H., Iwata, M. and Higuchi, T. (1997). "Gate-Level Evolvable Hardware: Empirical Study and Application." In *Evolutionary Algorithms in Engineering Applications*. Springer-Verlag: Berlin, 260-275.

12. Johnsonbaugh, R. (2001). Discrete Mathematics. Prentice Hall: New Jersey.

13. Kalganova, T. (2000). "Bidirectional Incremental Evolution in Extrinsic Evolvable Hardware." 2nd Workshop on Evolvable Hardware, 65.

14. Koza, J.R. (1992). Genetic Programming. MIT Press: Cambridge.

15. Koza, J.R., Bennett, F.H. III, Andre, D. and Keane, M.A. (1996). "Automated WYWIWYG Design of Both the Topology and Component Values of Electrical Circuits Using Genetic Programming." Genetic Programming 1996.

16. Koza, J.R., Bennett, F.H. III, Lohn, J., Dunlap, F., Keane, M.A. and Andre, D. (1997). "Automated Synthesis of Computational Circuits Using Genetic Programming." Evolutionary Computation 1997, 447-452.

17. Lang, K.J. (1995). "Hill Climbing Beats Genetic Search on a Boolean Circuit Synthesis Problem of Koza's." 12th Intl. Conf. on Machine Learning, No. 14.

18. Layzell, P. (1999). "Reducing Hardware Evolution's Dependency on FPGAs." 7th Intl. Conf. on Microelectronics for Neural, Fuzzy and Bio-Inspired Systems, 171.

19. Levi, D. and Guccione, S. (1999). "GeneticFPGA: Evolving Stable Circuits on Mainstream FPGA Devices." 1st Workshop on Evolvable Hardware, 12-17.

20. Li, W., Lim, A., Agrawal, P. and Sahni, S. (1992). "On The Circuit Implementation Problem." 29th Design Automation Conference, 478-483.

21. Lohn, J., Larchev, G. and DeMara, R. (2003). "A Genetic Representation for Evolutionary Fault Recovery in Virtex FPGAs." 5th Intl. Conf. on Evolvable Systems, 47-56.

22. Miller, J.F., Thompson, P. and Fogarty, T. (1997). "Designing Electronic Circuits Using Evolutionary Algorithms. Arithmetic Circuits: A Case Study." Genetic Algo. and Evol. Strategy in Eng. and Comp. Sci., 105-131.

23. Miller, J.F. (1999). "Evolution of Digital Filters Using a Gate Array Model." 1st Euro. Workshops on Evol. Image Analysis, Signal Proc. and Telecommun., 17-30.

24. Pinker, S. (1994). The Language Instinct. Perennial Classics: New York.

25. Pinker. S. (1999). Words and Rules. Weidenfeld & Nicolson: London.

26. Pfeifer, R. and Scheier, C. (2000). Understanding Intelligence. MIT Press: Massachusetts.

27. Reidys, C. and Stadler, P.F. (1998). "Neutrality in Fitness Landscapes." Submitted to Appl. Math. and Computation.

28. Russell, S. and Norvig, P. (2003). Artificial Intelligence: A Modern Approach. Prentice Hall: New Jersey.

29. Shang, L., Kaviani, A.S. and Bathala, K. (2002). "Dynamic Power Consumption in Virtex™-II FPGA Family." 10th Intl. Symp. on FPGAs, 157-164.

30. Thompson, A. (1995). "Evolving Fault Tolerant Systems." 1st Intl. Conf. on Genetic Algo. in Eng. Systems, 524-529.

31. Thompson, A. (1996). "Silicon Evolution." Genetic Programming 1996, 444-452.

32. Thompson, A. and Layzell, P. (1999). "Analysis of Unconventional Evolved Electronics." Comm. of ACM Vol. 42, 71-79.

33. Thompson, A., Layzell, P. and Zebulum, R.S. (1999). "Explorations in Design Space: Unconventional Electronics Design Through Artificial Evolution." Trans. in Evol. Computation, 167-196.

34. Tyrell, A.M., Hollingsworth, G. and Smith, S.L. (2001). "Evolutionary Strategies and Intrinsic Fault Tolerance." 3rd Workshop on Evol. Hardware, 98.

35. Vassilev, V.K., Job, D. and Miller, J.F. (2000). "Towards the Automatic Design of More Efficient Digital Circuits." 2nd Workshop on Evol. Hardware, 151.

36. Vassilev, V.K. and Miller, J.F. (2000). "Scalability Problems of Digital Circuit Evolution." 2nd Workshop on Evol. Hardware, 55.

37. Webb, B. (2002). "Robotics in Vertebrate Neuroscience." Nature 417, 359-363.

APPENDICES

APPENDIX A.

```
/*************************************************************************
****/
/*****                          *****/
/*****author: adam c. lammert *****/
/*****course: csc695           *****/
/*****begun: 21.august.2005    *****/
/*****                          *****/
/**********   circuit.c   ***********/


/**********/
/*****included files *****/
/**********/
#include <unistd.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <errno.h>
#include <sys/stat.h>
#include <sys/fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

/**********/
/*****constant definitions******/
/**********/
#define GENESIZE 78
#define TTROWS 16
#define GENERATIONSIZE 500
#define ARRAYHEIGHT 5
#define ARRAYWIDTH 5
#define NUMINPUTS 4
#define TOURNAMENTSIZE 8

/**********/
/*****variable declarations *****/
/**********/
int circuit [ARRAYHEIGHT][ARRAYWIDTH][3];  //holds a circuit for
simulation
int outputs [ARRAYHEIGHT][ARRAYWIDTH];  //outputs of each gate (for
simulation)
int final_gate [3];  //the final gate of the array
int function [GENERATIONSIZE];
int wires [GENERATIONSIZE];  //number of wires used by a given indiv.
int gates[GENERATIONSIZE];  //number of gates used by a given indiv.
int fitness [GENERATIONSIZE];  //fitness scores for each gene
int gene[GENERATIONSIZE][GENESIZE];  //each gene in current generation
int theOutputs[GENERATIONSIZE][TTROWS];
int children[GENERATIONSIZE][GENESIZE];  //children of current generation
int correct_output[] = {0,0,0,0, 0,0,0,0, 0,0,0,0, 0,0,0,1}; //minterms of
the specified function
int genebuf1 [GENESIZE];  //for crossover
int genebuf2 [GENESIZE];  //for crossover
int childbuf1 [GENESIZE];  //for crossover
```

```
int childbuf2 [GENESIZE];  //for crossover
int tournament_helper [TOURNAMENTSIZE];  //selection tournament helper
int tournament_temp [TOURNAMENTSIZE];  //selection tournament holder
int tournament [TOURNAMENTSIZE];  //selection tournament buffer
int input[NUMINPUTS];  //the specified inputs to the circuit (set below)
int used[ARRAYHEIGHT][ARRAYWIDTH];
float crand, worsttt, wires_score;
int pos, row, col;
int q, p, r, z, i, j, c;
int numAnd, numOr, numXor, numNot, numWire;
int bigloop ;
int gene1, gene2;
int worst, elite, zout, score, final_output, output;
int ouch, ouchnum;

/******/
/**MAIN METHOD**/
/******/
int main() {

  /***fill gene.txt with random genes***/
  srand((int)time(0));
  for(i=0;i<GENERATIONSIZE;i++) {
    for(j=0;j<GENESIZE;j++) {
      gene[i][j] = rand()%5;
    }
  }

  /***the master loop***/
  for(bigloop=0;bigloop<251;bigloop++) {

    /***setup for output.txt***/

    /***greater method for simulating circuit population****/
    for(zout=0;zout<GENERATIONSIZE;zout++) {

      /*initialize the wires array*/
      wires[zout]=0;

      /***read in the bulk of a gene***/
      p = 0;
      for(col=0;col<5;col++) {
        for(row=0;row<5;row++) {
          for(pos=0;pos<3;pos++) {
            c = gene[zout][p]; //extract the encoded gate specification
            circuit[col][row][pos] = gene[zout][p];
            p++;
            //what type of gate is specified?
            if(pos==2 && c==4) /*wire*/
              wires[zout]=wires[zout]+1; //count the wire
            else if(pos==2 && c==3) /*not*/
              wires[zout]=wires[zout]+0;
            else if(pos==2 && c==2) /*xor*/
              wires[zout]=wires[zout]+0;
            else if(pos==2 && c==1) /*or*/
              wires[zout]=wires[zout]+0;
            else if(pos==2 && c==0) /*and*/
```

```
        wires[zout]=wires[zout]+0;
    }
  }
}

/***read in final gate for the gene***/
for(pos=0;pos<3;pos++) {
  c = gene[zout][p];  //extract the encoded gate spec.
  final_gate[pos] = gene[zout][GENESIZE-3+pos];
  if(pos==2 && c==4) /*wire*/
    wires[zout]=wires[zout]+1;  //count the wire
  else if(pos==2 && c==3) /*not*/
    wires[zout]=wires[zout]+0;
  else if(pos==2 && c==2) /*xor*/
    wires[zout]=wires[zout]+0;
  else if(pos==2 && c==1) /*or*/
    wires[zout]=wires[zout]+0;
  else if(pos==2 && c==0) /*and*/
    wires[zout]=wires[zout]+0;
}

/***greater method for simulating one circuit***/
for(z=0;z<16;z++) {  //16 represents 2^n^n possible input configs.

  /***set the inputs to correct config***/
  if(z==0) {  //truth table row 0
    input[0] = 0;
    input[1] = 0;
    input[2] = 0;
    input[3] = 0;
  }
  if(z==1) {  //truth table row 1
    input[0] = 1;
    input[1] = 0;
    input[2] = 0;
    input[3] = 0;
  }
  if(z==2) {  //truth table row 2
    input[0] = 0;
    input[1] = 1;
    input[2] = 0;
    input[3] = 0;
  }
  if(z==3) {  //truth table row 3
    input[0] = 1;
    input[1] = 1;
    input[2] = 0;
    input[3] = 0;
  }
  if(z==4) {  //truth table row 4
    input[0] = 0;
    input[1] = 0;
    input[2] = 1;
    input[3] = 0;
  }
  if(z==5) {  //truth table row 5
    input[0] = 1;
```

```
  input[1] = 0;
  input[2] = 1;
  input[3] = 0;
}
if(z==6) {  //truth table row 6
  input[0] = 0;
  input[1] = 1;
  input[2] = 1;
  input[3] = 0;
}
if(z==7) {  //truth table row 7
  input[0] = 1;
  input[1] = 1;
  input[2] = 1;
  input[3] = 0;
}
if(z==8) {  //truth table row 8
  input[0] = 0;
  input[1] = 0;
  input[2] = 0;
  input[3] = 1;
}
if(z==9) {  //truth table row 9
  input[0] = 1;
  input[1] = 0;
  input[2] = 0;
  input[3] = 1;
}
if(z==10) {  //truth table row 10
  input[0] = 0;
  input[1] = 1;
  input[2] = 0;
  input[3] = 1;
}
if(z==11) { //truth table row 11
  input[0] = 1;
  input[1] = 1;
  input[2] = 0;
  input[3] = 1;
}
if(z==12) {  //truth table row 12
  input[0] = 0;
  input[1] = 0;
  input[2] = 1;
  input[3] = 1;
}
if(z==13) { //truth table row 13
  input[0] = 1;
  input[1] = 0;
  input[2] = 1;
  input[3] = 1;
}
if(z==14) { //truth table row 14
  input[0] = 0;
  input[1] = 1;
  input[2] = 1;
  input[3] = 1;
```

```
}
if(z==15) { //truth table row 15
  input[0] = 1;
  input[1] = 1;
  input[2] = 1;
  input[3] = 1;
}

/***simulate the first column***/
wires_score = 0;
for(row=0;row<5;row++) {
  circuit[0][row][0]=circuit[0][row][0]%4;//first col get inputs..
  circuit[0][row][1]=circuit[0][row][1]%4;//...from 1 thru 4
  if(circuit[0][row][2] == 0) //if encoding is 0, use AND operator
    outputs[0][row] =
    (input[circuit[0][row][0]] && input[circuit[0][row][1]]);
  if(circuit[0][row][2] == 1) //if encoding is 1, use OR operator
    outputs[0][row] =
    (input[circuit[0][row][0]] || input[circuit[0][row][1]]);
  if(circuit[0][row][2] == 2) { //if encoding is 2, XOR operator
    if(input[circuit[0][row][0]] == input[circuit[0][row][1]])
      outputs[0][row] = 0;
    else
      outputs[0][row] = 1;
  }
  if(circuit[0][row][2] == 3) //if encoding is 3, use NOT operator
    outputs[0][row] = !input[circuit[0][row][0]];
  if(circuit[0][row][2] == 4) {//if encoding is 4, WIRE operator
    outputs[0][row] = input[circuit[0][row][0]];
  }
}

/***simulate the intermediate columns***/
for(col=1;col<5;col++) {
  for(row=0;row<5;row++) {
    if(circuit[col][row][2] == 0) //if encoding is 0, AND operator
      outputs[col][row] =
        (outputs[col-1][circuit[col][row][0]] &&
         outputs[col-1][circuit[col][row][1]]);
    if(circuit[col][row][2] == 1) //if encoding is 1 OR operator
      outputs[col][row] =
        (outputs[col-1][circuit[col][row][0]] ||
         outputs[col-1][circuit[col][row][1]]);
    if(circuit[col][row][2] == 2) {//if encoding is 2 XOR operator
      if(outputs[col-1][circuit[col][row][0]] ==
         outputs[col-1][circuit[col][row][1]])
        outputs[col][row] = 0;
      else
        outputs[col][row] = 1;
    }
    if(circuit[col][row][2] == 3)  //if encoding is 3, use NOT
                                   //operator
      outputs[col][row] = !outputs[col-1][circuit[col][row][0]];
    if(circuit[col][row][2] == 4) {  //if encoding is 4, use WIRE
                                     //operator
      outputs[col][row] = outputs[col-1][circuit[col][row][0]];
    }
```

```
    }
  }

  /***simulate the final gate***/
  col=5;
  if(final_gate[2] == 0)  //if encoding is 0, use AND operator
    final_output =
      (outputs[col-1][final_gate[0]] &&
       outputs[col-1][final_gate[1]]);
  if(final_gate[2] == 1)  //if encoding is 1, use OR operator
    final_output =
      (outputs[col-1][final_gate[0]] ||
       outputs[col-1][final_gate[1]]);
  if(final_gate[2] == 2) {  //if encoding is 2, use XOR operator
    if(outputs[col-1][final_gate[0]] ==
       outputs[col-1][final_gate[1]])
      final_output = 0;
    else
      final_output = 1;
  }
  if(final_gate[2] == 3)  //if encoding is 3, use NOT operator
    final_output = !outputs[col-1][final_gate[0]];
  if(final_gate[2] == 4) {  //if encoding is 4, use WIRE operator
    final_output = outputs[col-1][final_gate[0]];
  }
  theOutputs[zout][z] = final_output;
}


/***determine num used gates in functional circuit***/
//the final gate is always used, so count it (if it's not a wire)
numWire = 0;
numAnd = 0;
if(final_gate[2] == 4) /*wire*/
  numWire++;
else
  numAnd++;
//reset the "used" array (keeps track of used gates)
for(col=0;col<5;col++) {
  for(row=0;row<5;row++) {
    used[col][row]=0;
  }
}
//which gates in row 5 are inputs to the final gate?
if(final_gate[2]>2)
  used[4][final_gate[0]]=1;
else {
  used[4][final_gate[0]]=1;
  used[4][final_gate[1]]=1;
}
//which gates in the previous row are used by the current row
//(recursive)?
for(col=0;col<4;col++) {
  for(row=0;row<5;row++) {
    if(circuit[4-col][row][2]>2 && used[4-col][row]==1) {
      used[4-col-1][circuit[4-col][row][0]]=1;
      if(circuit[4-col][row][2]==4)
```

```
          numWire++;
        else
          numAnd++;
      }
      else if(circuit[4-col][row][2]<3 && used[4-col][row]==1)
        {
          used[4-col-1][circuit[4-col][row][0]]=1;
          used[4-col-1][circuit[4-col][row][1]]=1;
          numAnd++;
        }
    }
  }
  col=0;
  //inspect the left-most column for contributing gates.
  for(row=0;row<5;row++) {
    if(used[col][row]==1) {
      if(circuit[col][row][2]==4)
        numWire++;
      else
        numAnd++;
    }
  }
  gates[zout] = numAnd;
}

/***greater method for functionality evaluation***/
for(i=0;i<GENERATIONSIZE;i++) {
  score = 0;
  for(z=0;z<16;z++) { //***how many minterms match up?
    if(theOutputs[i][z]==correct_output[z])
      score++;
  }
  /****new fitness eval****/
  if(score == (16/2)-1)
    score = 16/2;
  else if(score < (16/2)-1)
    score = 16-1-score;
  /****end newfit****/
  function[i]=score;
}

/***generate the fitness score for each parent***/
col=0;
worst=0;
worsttt=0.0;
wires_score=0.0;
for(i=0;i<500;i++) {  //for every gene in the generation
  if(function[i]<16) {
    fitness[i] = function[i]; //if functionality isn't perfect ...
                              //...size doesn't count in the score
  }
  else {  //if functionality is perfect, size must be counted
    //***used gates fitness
    fitness[i] = function[i] + 26 - gates[i];
    //***wire count fitness
    //fitness[i] = function[i] + wires[i];
    col++;
```

```
  }
  //gathering print-out information
  if(i==0)
  worst=fitness[i];
  worsttt=worsttt+(float)function[i];
  wires_score=wires_score+(float)wires[i];
}
//gathering print-out information
worsttt=worsttt/500.0;
wires_score=wires_score/500.0;
if(bigloop%50 == 0) {
  printf("\ngeneration:%d\n",bigloop);
  //printf("best.functionality:%d\n   ",worst);
  //printf("most.free.space:%d\n",26-gates[0]);
  //printf("num.gates.used:%d\n",gates[0]);
  if(col == 0)
    printf("no perfs yet!\n\n");
  else
    printf("num.gates.used:%d\n\n",gates[0]);
  //printf("average.functionality:%f\n",worsttt);
  //printf("average.unused.area:%f\n",wires_score);
  //printf("number.perfect.functions:%d\n\n",col);
}

// /***find the elite circuit, and move it on***/
worst=0;
elite=0;
for(i=1;i<GENERATIONSIZE;i++) {
  if(fitness[i]>fitness[elite])
    elite=i;
}
for(i=0;i<GENESIZE;i++) {
  genebuf1[i] = gene[elite][i];
  children[0][i] = genebuf1[i];
}

/***greater reproduction method***/
for(row=0;row<GENERATIONSIZE/2;row++) {
  for(q=0;q<2;q++) {
    for(i=0;i<TOURNAMENTSIZE;i++) {
      //randomly select some genes
      tournament_temp[i] = rand()%GENERATIONSIZE;
    }

    /*begin a tournament*/
    //sort the randomly selected genes in their array
    for(i=0;i<TOURNAMENTSIZE;i++)
      tournament_helper[i]=0;  //reset tournament helper
    for(zout=0;zout<TOURNAMENTSIZE;zout++) {
      c=0;
      while(tournament_helper[c]==1)
        c++;
      z=tournament_temp[c];
      for(i=c;i<TOURNAMENTSIZE;i++) {
        if((fitness[tournament_temp[i]]>fitness[z]) &&
           (tournament_helper[i]!=1)) {
          z=tournament_temp[i];
```

```
        worst = i;
      }
    }
    tournament[zout]=z;
    if(z==tournament_temp[c])
      tournament_helper[c]=1;
    else
      tournament_helper[worst]=1;
  }

  /*hold a tournament selection lottery*/
  //select a higher-fitness gene with
  //exponentially decreasing probability
  j=rand()%256;
  if(j==0)
    i=0;
  else if(j<2) //.00391 prob. of choosing #8
    i=7;
  else if(j<4) //.00781 prob. of choosing #7
    i=6;
  else if(j<8) //.01563 prob. of choosing #6
    i=5;
  else if(j<16) //.03125 prob. of choosing #5
    i=4;
  else if(j<32) //.0625 prob. of choosing #4
    i=3;
  else if(j<64) //.125 prob. of choosing #3
    i=2;
  else if(j<128) //.25 prob. of choosing #2
    i=1;
  else if(j<256) //.5 prob. of choosing #1
    i=0;

  /*make a tournament selection*/
  if(q==0) { //first-round selection
    //printf("selected gene=%d,",tournament[i]);
    for(r=0;r<GENESIZE;r++)
      genebuf1[r] = gene[tournament[i]][r];
  }
  if (q==1) { //second-round selection
    for(r=0;r<GENESIZE;r++)
      genebuf2[r] = gene[tournament[i]][r];
  }
}

/***crossover/mutate/output children***/
//randomly choose a cross-over point
j=(rand()%(GENESIZE-1))+1;
//perform the cross-over
for(p=0;p<j;p++) {
  childbuf1[p] = genebuf1[p];
  childbuf2[p] = genebuf2[p];
}
for(p=j;p<GENESIZE;p++) {
  childbuf1[p] = genebuf2[p];
  childbuf2[p] = genebuf1[p];
}
```

```
      //perform the mutation
      for(p=0;p<GENESIZE;p++) {
        if((rand()%(GENERATIONSIZE/2)) == 17)
          childbuf1[p] = (rand()%5);
      }
      for(p=0;p<GENESIZE;p++) {
        if((rand()%(GENERATIONSIZE/2)) == 17)
          childbuf2[p] = (rand()%5);
      }
      //place the children in their array
      for(i=0;i<GENESIZE;i++) {
        children[row+1][i] = childbuf1[i];
      }
      for(i=0;i<GENESIZE;i++) {
        children[(row+1+(GENERATIONSIZE/2))][i] = childbuf2[i];
      }
    }

    /***remove parent file/replace with child file***/
    for(i=0;i<GENERATIONSIZE;i++)
      {
        for(j=0;j<GENESIZE;j++)
          {
            gene[i][j] = children[i][j];
            children[i][j] = 0;
          }
      }
  }
}

/************************/
/***end of file***/
/************************/
/*************************************************************************
****/
```

APPENDIX B.

```
 /*****************************************/
/*****                           *****/
/*****author: adam c. lammert *****/
/*****course: csc695          *****/
/*****begun: 16.october.2005   *****/
/*****                           *****/
/**********   bfs.java   ***********/

import java.util.Vector;
import java.util.Hashtable;
import java.util.GregorianCalendar;

public class BFS
{
    private static int NOT = 3;
    private static int AND = 0;
    private static int OR = 1;
    private static int XOR = 2;
    private static int NONE = 99999;

    private static int numInputs = 4;
    private static int numOutputs = 16;
    private static int numTables = 65536;

    private static int firstBlank = 0;
    private static int firstNextLevel = 1;
    private static int firstNewTables = 1;
    private static int iterator1 = 0;
    private static int iterator2 = 1;
    private static int iteratorOp = 0;
    private static int i;
    private static int j;
    private static int k;
    private static int numConsidered = 0;
    private static int row;
    private static int col;
    private static int currentLevel = 0;

    private static int firstBlankTemp;
    private static int firstNextLevelTemp;
    private static int firstNewTablesTemp;
    private static int iterator1Temp;
    private static int iterator2Temp;
    private static int currentLevelTemp;
    private static long t1;
    private static long t2;

    private static int[] level = new int[65536];
    private static int[] prevOperator = new int[65536];
    private static int[] prevTable1 = new int[65536];
    private static int[] prevTable2 = new int[65536];
    private static int[][] tableLayout = new int[65536][16];
    private static int[] keys = new int[65536];
    private static int[] tableBuffer = new int[16];
```

```
private static int[] xTable = {0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1};
private static int[] yTable = {0,0,0,0,1,1,1,1,0,0,0,0,1,1,1,1};
private static int[] zTable = {0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1};
private static int[] wTable = {0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1};
private static int[] goalTable = {1,1,0,1,0,0,1,1,1,0,1,0,0,1,0,0};
private static Hashtable keysHash = new Hashtable();
private static int num;

//the MAIN method
public static void main(String args[])
{
    initializeKeys();
    initializeVariableLayouts();

    while(firstBlank < 65536) //proceed until all possibilities have
                              //been explored
        {
            System.out.println("level: " + currentLevel);
            resetAllIterators();  //reset iterators for a new phase
            //***store iterators for safekeeping
            firstNewTablesTemp = firstNewTables;
            iterator1Temp = iterator1;
            iterator2Temp = iterator2;
            firstNextLevelTemp = firstNextLevel;
            currentLevelTemp = currentLevel;
            //***begin new phase!
            if(currentLevel != 1)
                iterator1 = 0;
            //***consider combos with NOT first
            while(iterator1 < firstNextLevel)
                {
                    createNewCombo(3); //3 = NOT gate
                    iteratorOp = 0;
                    iterator1++;
                }
            //***restore iterators as they were above
            firstNewTables = firstNewTablesTemp;
            iterator1 = iterator1Temp;
            iterator2 = iterator2Temp;
            firstNextLevel = firstNextLevelTemp;
            currentLevel = currentLevelTemp;
            //***consider combos with all other gates
            while(iterator1 < (firstNextLevel - 1 ))
                {
                    //***create the new combos here
                    if((level[iterator1]+level[iterator2]) <
                                        currentLevel) {
                        for(i = 0; i < 3; i++) {
                            createNewCombo(i);
                        }
                    }
                    //***if no combos left in this phase, break
                    if((level[iterator1]==currentLevel-1) &&
                       (level[iterator2]==currentLevel-1))
                        break;
                    //***reset iterators for a new set of combos
                    resetLocalIterators();
```

```
                }
        }
}

//4***initialize the key array (i.e. the already-seen functions)
private static void initializeKeys() {
    for(i = 0 ; i < 65536 ; i++)
        keys[i] = 0;
}

//4***initialize the layout array
private static void initializeVariableLayouts() {
    //***set up the 'x' variable truth table column
    for(i = 0 ; i < 16 ; i++) {
        tableLayout[firstBlank][i] = xTable[i];
    }
    //***set up the 'x' variable book-keeping areas
    firstBlank++;
    level[0] = currentLevel;
    prevOperator[0] = NONE;
    prevTable1[0] = NONE;
    prevTable2[0] = NONE;
    //***set up the 'y' variable truth table column
    for(i = 0 ; i < 16 ; i++) {
        tableLayout[firstBlank][i] = yTable[i];
    }
    //***set up the 'y' variable book-keeping areas
    firstBlank++;
    level[1] = currentLevel;
    prevOperator[1] = NONE;
    prevTable1[1] = NONE;
    prevTable2[1] = NONE;
    //***set up the 'z' variable truth table column
    for(i = 0 ; i < 16 ; i++) {
        tableLayout[firstBlank][i] = zTable[i];
    }
    //***set up the 'z' variable book-keeping areas
    firstBlank++;
    level[2] = currentLevel;
    prevOperator[2] = NONE;
    prevTable1[2] = NONE;
    prevTable2[2] = NONE;
    //***set up the 'w' variable truth table column
    for(i = 0 ; i < 16 ; i++) {
        tableLayout[firstBlank][i] = wTable[i];
    }
    //***set up the 'w' variable book-keeping areas
    firstBlank++;
    level[3] = currentLevel;
    prevOperator[3] = NONE;
    prevTable1[3] = NONE;
    prevTable2[3] = NONE;
}

//4***add a key to the key table
//***this function takes a newly-produced function and generates
```

```
//***a key for it so it can be placed in the hash table
//***this is critical for checking whether succeeding functions
//***are redundant or not.
//***the key corresp. to the decimal value of the function's minterms
private static void addKey() {
    if(tableBuffer[0] == 1)  //is the most signif. minterm present?
        keys[firstBlank] = keys[firstBlank] + 32768; //if so, add its
                                                     //dec. value
    if(tableBuffer[1] == 1)
        keys[firstBlank] = keys[firstBlank] + 16384;
    if(tableBuffer[2] == 1)
        keys[firstBlank] = keys[firstBlank] + 8192;
    if(tableBuffer[3] == 1)
        keys[firstBlank] = keys[firstBlank] + 4096;
    if(tableBuffer[4] == 1)
        keys[firstBlank] = keys[firstBlank] + 2048;
    if(tableBuffer[5] == 1)
        keys[firstBlank] = keys[firstBlank] + 1024;
    if(tableBuffer[6] == 1)
        keys[firstBlank] = keys[firstBlank] + 512;
    if(tableBuffer[7] == 1)
        keys[firstBlank] = keys[firstBlank] + 256;
    if(tableBuffer[8] == 1)
        keys[firstBlank] = keys[firstBlank] + 128;
    if(tableBuffer[9] == 1)
        keys[firstBlank] = keys[firstBlank] + 64;
    if(tableBuffer[10] == 1)
        keys[firstBlank] = keys[firstBlank] + 32;
    if(tableBuffer[11] == 1)
        keys[firstBlank] = keys[firstBlank] + 16;
    if(tableBuffer[12] == 1)
        keys[firstBlank] = keys[firstBlank] + 8;
    if(tableBuffer[13] == 1)
        keys[firstBlank] = keys[firstBlank] + 4;
    if(tableBuffer[14] == 1)
        keys[firstBlank] = keys[firstBlank] + 2;
    if(tableBuffer[15] == 1)
        keys[firstBlank] = keys[firstBlank] + 1;
}


//4***reset iterators for a new phase
private static void resetAllIterators() {
    firstNewTables = firstNextLevel;
    iterator1 =  0; //go back...
    iterator2 =  1; //...to the beginning
    firstNextLevel = firstBlank;
    currentLevel++; //new phase is beginning
}

//4***reset for beginning a new set of combinations
private static void resetLocalIterators() {
    iteratorOp = 0; //reset operation iterator
    if(iterator2 == (firstNextLevel - 1)) { //if i2 is too big
        iterator1++; //increment i1
        iterator2 = iterator1+1; //reset i2
    }
```

```
    else  //increment i2
        iterator2++;
}

//4***check table buffer for goal state
private static int checkForGoal() {
    for(j = 0 ; j < 16 ; j++) {
        if(tableBuffer[j] != goalTable[j]) {
            return 0;
        }
    }
    return 1;
}

//4***check table buffer for redundant state
//***check the hashtable to see if the new function is actually old
private static int checkForRedundancy() {
    num++;
    if(num%1000==0)
        System.out.println("considered: " + num);
    addKey();
    if((Boolean)keysHash.containsKey((Integer)keys[firstBlank])) {
        keys[firstBlank] = 0;
        return 1;
    }
    else {
        keys[firstBlank] = 0;
        return 0;
    }
}

//4***if goal is found
//recursive function which prints out the right function
private static void endGame(int index) {
    if(index < 4) {} //if a variable is reached on the way up the tree
                     //do nothing, since that is the end of the line
    else {
        System.out.println("indx: " + index);
        //***is it an AND?
        if(prevOperator[index] == 0)
            System.out.println("   op: " +
                               prevTable1[index] +
                               " AND " +
                               prevTable2[index]);
        //***is it an OR
        else if(prevOperator[index] == 1)
            System.out.println("   op: " +
                               prevTable1[index] +
                               " OR " +
                               prevTable2[index]);
        //***is it an XOR?
        else if(prevOperator[index] == 2)
            System.out.println("   op: " +
                               prevTable1[index] +
                               " XOR " +
                               prevTable2[index]);
        //***is it a NOT?
```

```
        else if(prevOperator[index] == 3)
            System.out.println("   op: NOT " +
                               prevTable1[index]);
        //***what level is this one at?
        System.out.println("    lvl: " + level[index]);
        endGame(prevTable1[index]);
        if(prevOperator[index] != 3)  //recurse up the function tree
            endGame(prevTable2[index]);
    }
}

//4***create new combos
private static void createNewCombo(int op) {
    //***perform the correct operation on the truth table columns
    //***specified by iterator1 and iterator2
    for(row = 0 ; row < 16 ; row++) {
        //***if op = AND
        if(op == 0)
            tableBuffer[row] =
                ((tableLayout[iterator1][row]) *
                    (tableLayout[iterator2][row]))%2;
        //***if op = OR
        else if(op == 1)
            if((tableLayout[iterator1][row] == 1) &&
              (tableLayout[iterator2][row] == 1))
                tableBuffer[row] = 1;
            else
                tableBuffer[row] =
                    ((tableLayout[iterator1][row]) +
                          (tableLayout[iterator2][row]));
        //***if op = NOT
        else if(op == 3)
            tableBuffer[row] =
                ((tableLayout[iterator1][row] + 1)%2);
        //***if op = XOR
        else
            if((tableLayout[iterator1][row]) ==
              (tableLayout[iterator2][row]))
                tableBuffer[row] = 0;
            else tableBuffer[row] = 1;
    }
    numConsidered++;
    //***if a goal is found, print the results and get out!
    if(checkForGoal() == 1) {
        prevTable1[firstBlank] = iterator1;
        prevTable2[firstBlank] = iterator2;
        prevOperator[firstBlank] = op;

        if(op!=3)
            level[firstBlank] = level[iterator1]+level[iterator2]+1;
        else
            level[firstBlank] = level[iterator1]+1;
        System.out.println("nodes.considered: " + numConsidered);
        endGame(firstBlank);
        System.exit(-1);
    }
    //***if the function was not generated redundantly
```

```
            //***put it in the master table and include its book-keeping info
            else if((checkForRedundancy() != 1)) {
                prevTable1[firstBlank] = iterator1; //bookkeeping
                prevTable2[firstBlank] = iterator2; //bookkeeping
                prevOperator[firstBlank] = op; //bookkeeping
                if(op!=3) //bookkeeping
                    level[firstBlank] = level[iterator1]+level[iterator2]+1;
                else
                    level[firstBlank] = level[iterator1]+1;
                for(row = 0 ; row < 16 ; row++) { //add the new function
                    tableLayout[firstBlank][row] = tableBuffer[row];
                }
                //add the new function to the redundancies hash table
                addKey();
                keysHash.put(new Integer(keys[firstBlank]),new Integer(1));
                firstBlank++;
                if(firstBlank%10000 == 0)
                    System.out.println("firstblank = " + firstBlank);
            }
        }
    }
```