1 Introduction

In this project report, a methodology and results for the application of genetic programming to the minimization of combinatorial logic circuits is presented. The primary motivation for this project is the to address the difficulty in designing optimally sized, fully-functional combinatorial logic circuits for a large number of inputs using deterministic methods. However, the performance of the algorithm means that this motivation is not fully addressed, and instead a baseline for my future work is presented. One could consider $n > 15$ as a large number of inputs, $n \approx 10$ as a medium number of inputs, and $n \leq 4$ as a small number of inputs. Problems with 2, 3, and 4 inputs are tested, with good results for n=2,3 and poor results for n=4.

Combinatorial logic circuits are stateless, have no memory, and can be represented in the form of Boolean expressions. Such circuits consist of AND, OR, and NOT logic gates. The use of metaheuristic optimization algorithms for combinatorial logic circuit design is considered evolvable hardware. Previous works have utilized genetic algorithms [5, 6, 11] and Cartesian genetic programming [7] for the optimization of combinatorial logic circuits, with varied results. Evolvable hardware has particular applications for approximate computing, in which approximate circuits that trade full functionality for less resource usage [1] are designed.

This project report describes why genetic programming is chosen for optimization, describes the implementation of key optimization techniques used, and describes the algorithm implementation in detail. Combinatorial logic circuits can be represented as Boolean expressions, and these Boolean expressions can be represented in a binary tree data structure. Representing Boolean expressions as binary trees leads to the natural choice of genetic programming. Objective and fitness functions, constraints, reproduction operators, and other key parts of the implementation are described in detail. The base structure for the genetic programming algorithm is provided in [10], and modified herein for combinatorial logic circuit design. The fitness function defined in this report is a function of a candidate solution's size (number of gates) and correctness (number of correct outputs from the truth table). The key constraint is to only consider multiple-input, single-output circuits. The choice to only consider single-output problems reduces the difficulty of implementation, but significantly reduces the scope of problems that can be considered, which is a major drawback. Results are presented, with a discussion of what steps should be taken to improve the algorithm in order to attain better results and expand the scope of problems that can be considered.

Overall, while this report provides a good baseline for my future work, it fails to address the primary motivation. It also led to a greater understanding of how my approach might be applicable to approximate computing, as this appears to be a more achievable paradigm to operate within, rather than fully-functional circuits. While results are promising for small problem sizes in terms of walltime and generated circuits, it is clear than improvements must be made before attempting to optimize large circuits.

2 Background

This section describes background information relevant to understanding the design of combinatorial logic circuits.

Combinatorial logic circuits

Combinatorial logic circuits are circuits consisting of logic gates that can be represented with Boolean algebraic operators, and transform digital inputs to digital outputs. Combinatorial logic circuits are stateless and have no memory, meaning outputs depend entirely on present input values. The minimization of combinational logic circuits is important to designers who wish to save space, and therefore cost. However, creating an efficient design is a difficult task when there are more than a small number of variables [4]. The creation of combinatorial logic circuits typically begins with the construction of a truth table, which shows how different combinations of inputs provide the desired output [9]. An example of such a truth table is shown in Table 1. A truth table could be generated from a non-minimized boolean expression, or a logical expression. From this truth table, a minimized boolean representation of the logic circuit can be found, and then implemented using AND, OR, and NOT logic gates. Therefore, the design of combinatorial logic circuits is equivalent to the design of Boolean expressions. Figure 1 shows a minimized logic circuit implementation of the Boolean expression $(\overline{A} \cap B) \cup (A \cap \overline{B})$, which represents the XOR(A,B) logic operation.

Table 1: Truth table for 2-input XOR gate

A	B	XOR(A,B)
0	0	0
0	1	1
1	0	1
1	1	0

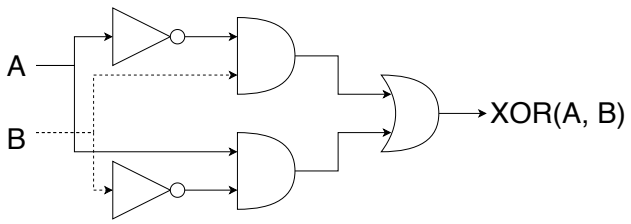

Figure 1: Minimized logic circuit for XOR(A,B) as defined in Table 1

Deterministic methods for minimization of Boolean expressions

For smaller problems, deterministic algorithms such as Karnaugh Maps and the Quine-McCluskey method are used, but they do not scale well beyond problem sizes $n = 4$ and $n = 10$ respectively [3]. The Quine-McCluskey method is also known as the method of

prime implicants. There are two steps to this method; first generate all prime implicants, then find the minimum coverage [3]. Karnaugh Maps are a manual method of solving small problems (n=2 to n=4), while the Quine-McCluskey method is an algorithm that can be programmed for any number of input variables. However, the main limitation of the Quine-McCluskey method is that the number of prime implicants generated by an expression grows exponentially with input size [3], meaning it has very high computational complexity. The high computational complexity leads to very long runtimes with increasing problem sizes, with even medium sized problems (n=10 to n=15) posing challenges [3].

3 Related Work

Heuristic optimization for combinatorial logic circuit design

Several potentially non-optimal heuristic methods have been proposed for problems with large n, as problems can have sizes far exceeding $n = 15$. One example is ESPRESSO-II, which achieves both robust performance and quality results [2]. However, such methods still can fall victim to non-optimal convergence, despite their improved complexity and lower resource requirements [4].

Metaheuristic optimization for combinatorial logic circuit design

Another set of approaches for combinatorial logic circuit minimization includes evolvable hardware, in which evolutionary algorithms are applied for the minimization of Boolean expressions. Evolvable hardware is made possible by the use of re-programmable hardware, such as Field Gate Programmable Arrays (FPGA) [11]. For example, work has been done to implement genetic algorithms specifically for bit-adder design [6], with results showing 100% functional circuits for three and four-bit adders. With programmable logic devices (PLD), the general approach is to represent the PLD architecture bits as the genetic algorithm chromosomes [5]. In [5], genetic algorithm results vary; for example a 6 channel multiplexer is evolved with fairly high performance, while results with a 4 channel parity checker showed only moderate performance. In all of these approaches, some correct or similar Boolean expression is used as the starting point for the genetic algorithm.

An extended application of evolvable hardware is approximate circuits. Approximate circuits are part of the approximate computing paradigm, in which lower resource consumption is required, but allows for less than 100% functional performance [1]. In [7], a Cartesian genetic programming approach is proposed, which allows for the improvement of evolved circuits and reduces the time for evolution. This Cartesian genetic programming approach is particularly relevant to my report, as I have chosen to use genetic programming as my metaheuristic programming approach. In [7], it is also made clear that the evolution of non-trivial circuits is not an easy task by any means. In [8], a systematic methodology for approximate circuit design is proposed, allowing designers to select from a set of proposed approximate solutions.

4 Methodology

This section describes the approach and implementation of genetic programming for Boolean expressions. The reasoning for using a genetic programming approach is described. Then, the node class, tree class, and search algorithm are described in detail.

Overview of approach

The approach taken in designing combinatorial logic circuits can be summarized as follows: given a truth table for some problem, generate Boolean expressions which may represent (solve) the problem. Boolean expressions were randomly generated, with no prior knowledge of what a good solution might look like, and then a metaheuristic search algorithm was used to optimize the size and correctness of generated expressions. This differs from what was described in the *Related Work* section, where the approaches are essentially to take an existing circuit and evolve it, in order to improve it. My approach requires little-to-none prior knowledge of the structure of the combinatorial circuits, but as will be shown in the *Results* section, this approach also has its drawbacks.

Determining objective functions and fitness function The problems of interest are multi-objective; minimize size, and maximize correctness (i.e. minimize error). To minimize size, an objective function was defined as follows:

$$size_i = f(candidate_i) = \text{count(number of \{AND, OR, NOT\} gates in candidate)}$$

To minimize error, an objective function to maximize correctness was defined as follows:

$$correctness_i = f(candidate_i) = \text{count(number of correct outputs given by candidate)}$$

Fitness was determined using a modified weighted function of the two objective functions. The weights w_{size}, $w_{correctness}$ for each objective function are tunable, in an attempt to accomodate different problem sizes. The fitness function was defined as follows:

$$fitness_i = (w_{size} \cdot \frac{size_i}{size_{max}}) + (w_{correctness} \cdot (1 - correctness_i))$$

A smaller fitness indicates a better candidate. The maximum number of gates $size_{max}$ was constant across all iterations, and depended on the maximum depth $\hat{d}$ hyperparameter. This will become clearer when maximum depth $\hat{d}$ of the trees in genetic programming is discussed later in this report.

Determining constraints Constraints for this optimization approach limit the types of combinatorial logic circuits that can be solved, but simplify the approach. The first constraint is that each problem can have any number of inputs, but must only have one output. The second constraint is that each logic gate can only have a maximum of two inputs. The third constraint is that only AND, OR, and NOT logic gates can be used, in order to properly represent Boolean expressions in a logic circuit. This third constraint means that

more advanced gates such as NAND or XOR are not used. However, such gates could be discovered in the optimization process, and the encapsulation reproduction operator may allow this discovery to be propagated through a member's offspring. While this idea might sound promising, the encapuslation operator has not been implemented yet, and so this idea has not been verified. All listed constraints were built into the design of the algorithm, rather than being constraints that limit the search space during the optimization process.

Deciding to use genetic programming

With the focus of combinational logic circuit design being the minimization of Boolean expressions, the first step was to represent Boolean expressions in a data structure for programming. Given that the problems of interest in this study are multiple-input, single-output problems, a tree data structure was a reasonable choice. However, this does not mean that multiple-output problems could not be represented with a tree data structure. Furthermore, with each AND and OR gate having two inputs and a single output, a binary tree representation was chosen. In order to accomodate NOT gates, it was decided that NOT gates would be represented as a weight in each node. The weight representing NOT gates tells us whether or not an output is complemented. Figure 2 shows how the Boolean expression $(\overline{A} \cap B) \cup (A \cap \overline{B})$ would be represented as a binary tree. While Figure 2 shows the complement on the tree links, in reality this was a node weight, as previously described.

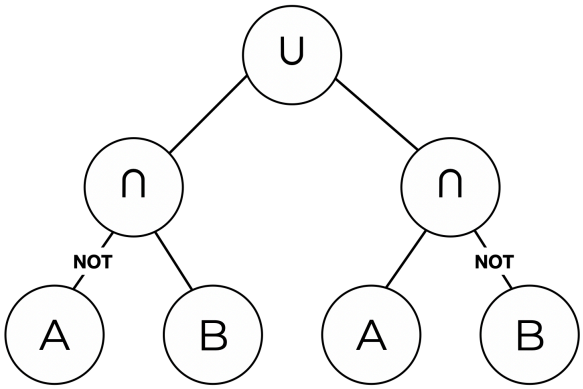

Figure 2: Binary tree representation of the logic circuit shown in Figure 1

With the binary tree representation formulated for Boolean expressions, the natural choice for a metaheuristic optimization approach was genetic programming. Genetic programming is the application of genetic algorithms to any tree data structure [10]. With the choice of metaheuristic optimization approach decided, the data structures and operations were then implemented using Python.

Implementing the *Node* class

The first step in creating the search algorithm was to create a *Node* class. The *Node* class required a few properties: type, weight, left child, and right child. The type is either a function type, or an input type. Function types were either an AND gate, or an OR gate. Each gate had to be implemented in Python, with 'Do Not Care' values from the truth table handled properly. 'Do Not Care' values were encoded with a string value of 'X'. For example, an AND gate with inputs of 0 and 'X' should always output 0. The input types were decided when the search algorithm was performed, as they depended on the truth table inputs. For example, a truth table with 2 inputs, 'A' and 'B' would mean the *Node* class could have either of these as its input type.

With the properties of the *Node* class defined, a single method was implemented, called *evaluate*. For nodes with a function type, the *evaluate* method would find the values supplied by each of the node's children, and then perform the function with both inputs. After applying the function, the *evaluate* method would use the node's weight to determine whether to complement the output value. For nodes with an input type, the *evaluate* method would return the value of the input in the truth table (for a given truth table row), and then complement it if required.

Implementing the *Tree* class

The *Tree* class was implemented for representing each genetic program. The properties of the *Tree* class included: root node, maximum depth $\hat{d}$, a list of available function types, and a list of available terminal types. The maximum depth was required in order to limit the depth of each program and avoid creating programs that were much larger than any reasonable solution. The list of available function types were always the AND and OR gates, while the list of terminal types were the input types for a given problem. When initializing a *Tree*, a root node was assigned by applying a tree creation method. The tree creation methods included *create_gp_full*, *create_gp_grow*, and *create_gp_ramped*.

Each creation method will be outlined now, and were provided by T. Weise in [10]. The *create_gp_full* method creates a tree that filled all nodes to the maximum depth. The *create_gp_grow* method creates a tree that randomly decided to keep growing (to max depth) or assign a terminal node. The *create_gp_ramped* method is the ramped-half-and-half implementation. In the ramped-half-and-half method, each tree randomly decides to create itself with the *create_gp_full* or *create_gp_grow* method. Using a ramped method is helpful, as both methods are applied with equal probability. In each creation method, the root node recursively generates its children using the given creation method.

In order to generate new candidate solutions, reproduction methods were implemented. The two reproduction methods were the *mutate_gp* and *recombine* methods. The *mutate_gp* method mutated a tree by randomly selecting a non-terminal node in the tree, and then creating a new subtree at this node. The *recombine* method first randomly selected a random node from itself. Then, either the left child or right child of the selected node was replaced

with a subtree from another tree in the population. In each reproduction method implementation, it was ensured that reproduction did not create new trees with depth greater than the defined maximum depth. While other reproduction methods such as permutation and encapsulation could have proved useful, they were not implemented in this project.

The other *Tree* class methods worth mentioning include *print_tree*, *evaluate*, and *find_size*. The *print_tree* method recursively printed out the tree in the form of a Boolean expression, so that results could be interpreted. The *evaluate* method was a wrapper method for the *Node* class *evaluate* method. The *Tree evaluate* method simply called the root node's *evaluate* method, using a given input pattern from the truth table, and then compared it to the correct outputs. A Boolean value was returned from the *Tree evaluate* method, indicating if the correct output was attained. The *find_size* method found the size of a given tree, by counting the number of nodes with a function type (AND, OR gates) as well as the number of complements (NOT gates). The *evaluate* and *find_size* methods were important for determining the fitness of candidate solutions.

Implementing the search algorithm

Algorithm 1: Genetic programming search

Input: T: The truth table for specified problem
Input: $\hat{d}$: the maximum tree depth
Input: *size_pop*: size of population
Input: *size_mate*: size of mating pool
Input: *num_gens*: number of generations
Input: *p_mutate*: probability of tree mutation
Input: *p_crossover*: probability of subtree crossover (recombination)
Input: *w_size*: weight for size in fitness function
Input: *w_correctness*: weight for correctness in fitness function
Output: g: the best solution from final population
begin
    create initial population using T;
    for *num_gens* **do**
        evaluate population correctnesses;
        evaluate population sizes;
        assign population fitnesses;
        select mating pool;
        perform reproduction with mating pool;
    end
    extract best member from final population;
    return g;
end

With the *Node* and *Tree* classes implemented, the search algorithm could be implemented. The general structure for the search algorithm was provided by T. Weise in [10], and is similar to general genetic algorithm approaches. A high-level representation of the genetic programming search is shown in Algorithm 1. As previously described, the evaluation of the population was done based on correctness and size of candidate solutions. From the correctness and size values, each member of the population was assigned a fitness using a weighted objective function. In every problem that was tested, a higher weighting was given to correctness. While a Pareto ranking fitness approach may have proved better, it was not implemented. The mating pool was selected using tournament selection. Reproduction was done by applying the mutation and recombination operators with defined probabilities to each member of the mating pool, until a new, full population was generated. The best member of the final member was extracted by calculating the fitnesses of the final population, and the member with the highest fitness was selected. If two or more solutions had equivalent best fitness, then the first best member found in the population was taken.

5 Results

In this section, significant results are presented.

Applying the search algorithm to 2-input (trivial) problems

For a baseline implementation, the search algorithm was tested on simple 2-input problems. The goal of testing these problems was to see how consistently the search algorithm would generate the smallest possible implementations of more advanced logic gates, such as XOR and NAND. For each of the tested problems, the best found hyperparameter tunings ended up being the same, and can be seen in Table 2. The results for these problems are summarized in Table 3, which show how the algorithm performed on each problem over 10 runs. The best solution was known before testing, and used to compare the results of the algorithm with the best possible implementations. The best solution is that with the minimum number of gates and 100% correctness. The performance of the algorithm for generating XOR, NAND, and XNOR problems were good, as the best solution was found in some runs. In other runs for these problems, a viable solution with a larger number of gates was found. However, for the XNOR problem, only 7 of the 10 runs yielded a solution that wa 100% correct. This means that although the algorithm was usually successful in solving these problems, it did not guarantee convergence to the global optimum.

Table 2: Hyperparameter settings for 2-input problems

max_depth	$size_{pop}$	$size_{mate}$	num_gens	p_{mutate}	$p_{recombine}$	$w_{correct}$	w_{size}
4	100	20	100	0.5	0.5	2	0.1

The best solution for the XOR problem was $((A \cap B)' \cap (A \cup B))$, with 4 gates and 100% correctness. Full results for the XOR problem can be seen in the Appendix, in Table 8. The best solution for the NAND problem was $(B \cap A)'$, with 2 gates and 100% correctness. Full

Table 3: Results for 2-input problems

Problem	Found best solution (/10 runs)	Found 100% correct solution (/10 runs)	Avg. walltime (seconds)
XOR	3	10	1.80
NAND	5	10	1.25
NOR	4	10	1.18
XNOR	3	7	1.62

results for the NAND problem can be seen in the Appendix, in Table 9. The best solution for the NOR problem was $(A \cup B)'$, with 2 gates and 100% correctness. Full results for the NOR problem can be seen in the Appendix, in Table 10. The best solution for the XNOR problem was $((B \cup A)' \cup (B \cap A))$, with 4 gates and 100% correctness. Full results for the XNOR problem can be seen in the Appendix, in Table 11.

Applying the search algorithm to a 3-input problem

In order to further test the applicability of the genetic programming approach, a simple 3-input problem was tested. The problem was defined and tested in two seperate ways: first, using 'Do Not Care' (i.e. 'X') values in the truth table, and then with all rows expanded. The key difference in the way this problem was defined is that using 'X' led to having 4 rows to check outputs on, while expanding the rows meant there were 9 rows to check outputs. Interestingly, vastly different results were found based on how the problem was defined, and can be seen in Table 5. The hyperparameter tunings for the 3-input problem can be seen in Table 4.

Table 4: Hyperparameter settings for 3-input problem

max_depth	$size_{pop}$	$size_{mate}$	num_gens	p_{mutate}	$p_{recombine}$	$w_{correct}$	w_{size}
4	100	10	100	0.5	0.5	1	0.01

Table 5: Results for 3-input problem

Problem	Found best solution (/10 runs)	Found 100% correct solution (/10 runs)	Avg. walltime (seconds)
Without 'X'	3	10	2.01
With 'X'	1	4	1.41

In either case, the best solution found was $((A \cup B) \cap (A \cap C)')$, with 4 gates and 100% correctness. The difference in performance will be explored further in the *Discussion* section of this report. Full results for this problem using 'X' values can be seen in the Appendix in Table 13. Full results for this problem not using 'X' values can be seen in the Appendix in Table 12.

Applying the search algorithm to a 4-input problem

To further test the genetic programming approach, it was applied to a (slightly more advanced) 4-input problem. The problem was set up without 'X' values, instead explicitly defining all 16 rows for the truth table. The hyperparameter tuning can be seen in Table 6. The results for this problem can be seen in Table 7. A fully correct solution was not found for this problem using the genetic programming approach. However, in several cases, a solution with 94% correctness was found. Each of these solutions varied in number of gates, with one having 14 gates, and another having 19 gates. The best solution found by the algorithm was:

$$(((C \cup D) \cap ((A \cup B) \cap ((D \cap C) \cup (B \cap A))))' \cap (((B \cup D) \cap A) \cup ((B \cap D) \cup C)))$$

Other solutions that the algorithm converged to most often had a correctness of 81% (3 runs) or 88% (3 runs). In the worst case, the correctness was 69% (1 run). The median number of gates found for this problem was 14.

Table 6: Hyperparameter settings for 4-input problem

max_depth	$size_{pop}$	$size_{mate}$	num_gens	p_{mutate}	$p_{recombine}$	$w_{correct}$	w_{size}
6	100	10	100	0.5	0.5	1	0.001

Table 7: Results for 4-input problem

Found best solution (/10 runs)	Found 100% correct solution (/10 runs)	Avg. walltime (seconds)
0	0	5.66

These results are somewhat dissapointing. It is also noted that the walltimes increase as problem sizes increase, even with the same number of generations, population size, etc. Other hyperparameter tunings were tried, namely increasing the number of generations, size of population, and size of mating pool. These other tunings caused longer walltimes without a noticable increase in quality of results. This indicates there may be improvements to how the search algorithm was implemented, for example changing the fitness function. The *Discussion* section of this report will delve deeper into what might have caused these results.

Ignoring higher order problems (e.g. n=10) for now

Given the lacklustre results for the 3-input and 4-input problems, the algorithm was not tested on higher order problems. While this means the original motivation of solving high order problems is not addressed in this report, the results thus far indicate the algorithm should be improved before moving onto bigger problems. Further discussion of all of the results and their implications follow in the *Discussion* section.

6 Discussion

This section will focus on explaining and understanding the results presented in the *Results* section, as well as potential improvements to the methodology. There are several key focuses in this discussion. First, it should be understood why the algorithm did not always converge to the global optimum. Second, there are concerns regarding differences in performance depending on how the problem was defined in the truth table (with and without 'X' values). Finally, the failure of the algorithm to find a fully viable solution to the 4-input problem should be understood. A discussion of these problems will lead to suggestions on how the algorithm might be improved, or what limitations are inherent to my approach. The benefits of my approach will also be discussed, as the algorithm did work to some extent in its current form. There are additional limitations in the design of the algorithm that limit what kind of problems can be solved, and this will also be explored further.

Understanding why the search algorithm failed in certain cases

The search algorithm failed to converge to a global optimum in some of the problems, and failed to find a single correct solution for the 4-input problem. The first, most obvious potential source of this problem is the inherent randomness in metaheuristic optimization techniques, specifically with genetic algorithms/programming. The search algorithm did not search the entire search space, and had no knowledge of what a good solution might look like, aside from being guided by previous discoveries. Therefore, there are no guarantees it will converge to a global optimum. The randomness also explains why the algorithm converged to different correct solutions as well. However, the randomness alone is not enough to explain why the algorithm did not coverge for the 4-input problem. Perhaps the fitness function is not robust enough. Or, it could be helpful to maximize exploration in early iterations, and then maximize exploitation in later iterations; currently the exploration/exploitation trade-off is constant throughout the search. This dynamically changing trade-off could be achieved with dynamic reproduction rates. Another idea that could help is the use of a 'hall-of-fame', to ensure the best solution at each step is retained throughout the search. With the aforementioned changes, an increase in the number of generations may also help find a better solution. Overall, more work needs to be done to improve the search algorithm, before moving on to larger problem sizes.

With regards to the 3-input problem, it is strange that defining the truth table with or without 'X' values changed the results. It is possible that with more rows to check outputs with, the algorithm could better distinguish different candidate solutions. An 'X' value in a row does not necessarily mean that the designed circuit will output the same result for either a 0 or 1 in the place of that 'X'. More investigation is required to understand why results change for the same problem, depending on how the truth table is constructed.

Benefits of genetic programming approach

A few benefits of the genetic programming approach will be discussed. First, the approach retained relatively short walltimes with increasing problem size (for tested problems); which

is usually a big problem for designing combinatorial circuits. Secondly, over 10 runs, one could easily pick out the best found solution, meaning despite the randomness, a good solution was often found in a reasonable amount of seperate runs. Finally, this approach was fairly robust on small problem sizes and required no prior knowledge of what a good design might be, meaning this approach could be beneficial in cases where the ideal circuit is not already known. But, as previously discussed, improvements must be made before such problems can be tested, as these problems would be of higher order. It could be beneficial to start with potential solutions to a problem and evolve them as discussed in the *Related Work* section, rather than with a set of completely random candidate solutions. So, even the benefit of less required prior knowledge itself poses issues in how well the algorithm performs.

Limitations of this genetic programming implementation

Some limitations in the design of the search algorithm will be discussed here. First, the constraints in the design of the algorithm limit which problems can be solved. Many combinatorial logic circuits require multiple outputs, which is not allowed in this approach. A simple example of such a problem would be a bit-adder with sum and carry outputs. A potential solution to this would be to check the output patterns at every node of the tree, rather than just the root node. By checking all output patterns, it could be seen if the required output patterns exist in the circuit, rather than if the entire circuit produces a single output of interest.

Another limitation of the design is the use of binary trees. For evaluation and other methods, the tree had to be searched, and in the worst case this had computational complexity of $O(n)$. Clearly, as problem size increases, this could pose a problem. Perhaps genetic algorithms are more beneficial in these cases, as larger strings could be less computationally complex to evaluate than larger trees.

Potential applications to approximate circuits

If similar results for the 4-input problem were found for higher order problems, this approach could have implications for approximate circuits. The design of approximate circuits is concerned with producing circuits that might contain some error, but use less resources [1]. For example, if a circuit with 94% correctness was discovered with the search algorithm for a higher order problem, it could prove useful for approximate circuits. However, improvements such as removing the single-output constraint and improving the algorithm performance should be made before considering this application.

7 Conclusion

Overall, while the original motivation for this project was not fully addressed, some inroads have been made on the benefits and limitations of applying genetic programming to combinatorial logic circuit design. Combinatorial logic circuits are those that can be represented with Boolean expressions, and the design is typically done with methods such as

the ESPRESSO-II algorithm. An approach for solving such problems was developed using binary trees to represent Boolean expressions, with genetic programming implemented to generate and optimize such trees. Problems with 2, 3, and 4 inputs and a single output were tested, finding that the algorithm in its current state is good at solving problems with 2 or 3 inputs. The constraints built into the design limited the performance and applicability. Performance was lessened by not starting with a circuit that could reasonably solve the problems, and instead using randomly generated candidates. Applicability was lessened by constraining the problem space to only single-ouput problems.

It is promising that my implementation of genetic programming provided good results for combinatorial logic circuit problems with very few inputs. While the results are not satisfactory beyond n=3, they provide a baseline from which I am able to make improvements beyond the scope of this project. In order to improve the applicability of my approach, the single-output constraint should be removed, by changing the design so that multiple-output problems can be considered. Furthermore, other improvements to the algorithm could be implemented, such as improving the fitness function, or dynamically changing the exploration-exploitation trade-off. With further work, it could be possible to design a genetic programming search algorithm that is robust enough to handle problems with large input sizes in some reasonable amount of time. In conclusion, this project has provided me with a decent baseline implementation of genetic programming for combinatorial logic circuit design, and the understanding to further explore the application of this approach to approximate circuits.

References

[1] Alberto Bosio et al. "Design, Verification, Test and In-Field Implications of Approximate Computing Systems". In: *25th IEEE European Test Symposium*. Los Alamitos, US: Institute of Electrical and Electronics Engineers, 2020, pp. 1–10. ISBN: 978-1-7281-4312-5. DOI: 10.1109/ETS48528.2020.9131557. URL: https://www.fit.vut.cz/research/publication/12228.

[2] Robert K. Brayton et al. "Comparisons and Conclusions". In: *Logic Minimization Algorithms for VLSI Synthesis*. Boston, MA: Springer US, 1984, pp. 160–173. ISBN: 978-1-4613-2821-6. DOI: 10.1007/978-1-4613-2821-6_7. URL: https://doi.org/10.1007/978-1-4613-2821-6_7.

[3] Robert K. Brayton et al. "Introduction". In: *Logic Minimization Algorithms for VLSI Synthesis*. Boston, MA: Springer US, 1984, pp. 1–14. ISBN: 978-1-4613-2821-6. DOI: 10.1007/978-1-4613-2821-6_1. URL: https://doi.org/10.1007/978-1-4613-2821-6_1.

[4] Lucas Augusto Müller de Souza et al. "A benchmark suite for designing combinational logic circuits via metaheuristics". In: *Applied Soft Computing* 91 (2020), p. 106246. ISSN: 1568-4946. DOI: https://doi.org/10.1016/j.asoc.2020.106246. URL: http://www.sciencedirect.com/science/article/pii/S1568494620301861.

[5] Hitoshi Iba, Masaya Iwata, and Tetsuya Higuchi. "Machine learning approach to gate-level Evolvable Hardware". In: *Evolvable Systems: From Biology to Hardware*. Ed. by Tetsuya Higuchi, Masaya Iwata, and Weixin Liu. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 327–343. ISBN: 978-3-540-69204-1.

[6] Julian Miller et al. "Designing Electronic Circuits Using Evolutionary Algorithms. Arithmetic Circuits: A Case Study". In: *Genetic Algorithms and Evolution Strategies in Engineering and Computer Science* (Oct. 1999).

[7] Z. Vasicek and L. Sekanina. "Evolutionary Approach to Approximate Digital Circuits Design". In: *IEEE Transactions on Evolutionary Computation* 19.3 (2015), pp. 432–444. DOI: 10.1109/TEVC.2014.2336175.

[8] R. Venkatesan et al. "MACACO: Modeling and analysis of circuits for approximate computing". In: *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2011, pp. 667–673. DOI: 10.1109/ICCAD.2011.6105401.

[9] Christopher Vickery. *Minimizing Boolean Functions*. 2011. URL: https://babbage.cs.qc.cuny.edu/courses/Minimize/.

[10] Thomas Weise. *Global Optimization Algorithms. Theory and Applications*. Third. Dec. 2011.

[11] X. Yao and T. Higuchi. "Promises and challenges of evolvable hardware". In: *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 29.1 (1999), pp. 87–97. DOI: 10.1109/5326.740672.

Appendix: Full tables of results

Table 8: Full results for XOR problem

Run #	Solution	Correctness	Size	Walltime (s)
1	$((A \cap B)' \cap (A \cup B))$	100%	4	1.81
2	$((A \cap B') \cup (B \cap A'))$	100%	5	1.80
3	$((B' \cap A) \cup (B \cap A'))$	100%	5	1.77
4	$((A \cap B)' \cap (B \cup A))$	100%	4	1.80
5	$((A' \cap B) \cup (B' \cap A))$	100%	5	1.81
6	$((A \cup B') \cap (A' \cup B))'$	100%	6	1.85
7	$((A \cap B') \cup (A' \cap B))$	100%	5	1.78
8	$((B' \cup A)' \cup (B' \cap A))$	100%	6	1.81
9	$((A \cap B) \cup (B \cup A)')'$	100%	5	1.78
10	$((A \cup B) \cap (A \cap B)')$	100%	4	1.80

Table 9: Full results for NAND problem

Run #	Solution	Correctness	Size	Walltime (s)
1	$(B' \cup A')$	100%	3	1.18
2	$(A' \cup B')$	100%	3	1.21
3	$(B \cap A)'$	100%	2	1.16
4	$(B' \cup A')$	100%	3	1.21
5	$(A' \cup B')$	100%	3	1.20
6	$(A \cap B)'$	100%	2	1.15
7	$((B \cap A)' \cap (B \cap A)')$	100%	5	1.79
8	$(B \cap A)'$	100%	2	1.19
9	$(A \cap B)'$	100%	2	1.18
10	$(A \cap B)'$	100%	2	1.19

Table 10: Full results for NOR problem

Run #	Solution	Correctness	Size	Walltime (s)
1	$(A' \cap B')$	100%	3	1.18
2	$(A \cup B)'$	100%	2	1.15
3	$(A \cup B)'$	100%	2	1.15
4	$(A \cup B)'$	100%	2	1.20
5	$(A' \cap B')$	100%	3	1.23
6	$(A' \cap B')$	100%	3	1.16
7	$(A' \cap B')$	100%	3	1.16
8	$(A' \cap B')$	100%	3	1.19
9	$(A \cup B)'$	100%	2	1.17
10	$(B' \cap A')$	100%	3	1.24

Table 11: Full results for XNOR problem

Run #	Solution	Correctness	Size	Walltime (s)
1	$((B \cup A)' \cup (B \cap A))$	100%	4	1.82
2	$((A \cup B') \cap (A' \cup B))$	100%	5	1.83
3	$(A \cap B)$	75%	1	1.20
4	$((A' \cup B) \cap (A \cup B'))$	100%	5	1.79
5	$((A \cap B) \cup (B \cup A)')$	100%	4	1.76
6	$((B \cup A') \cap (B' \cup A))$	100%	5	1.78
7	$(A \cap B)$	75%	1	1.18
8	$((B' \cup A) \cap (A' \cup B))$	100%	5	1.84
9	$((A \cap B) \cup (A \cup B)')$	100%	4	1.79
10	$(B \cap A)$	75%	1	1.18

Table 12: Full results for 3-input problem (not using 'Do Not Care' in truth table)

Run #	Solution	Correctness	Size	Walltime (s)
1	$((A' \cup C') \cap (A \cup B))$	100%	5	1.89
2	$((A \cup B) \cap (A \cap C)')$	100%	4	1.94
3	$((B \cup A) \cap (A \cap C)')$	100%	4	1.96
4	$((C' \cup A') \cap (A \cup B))$	100%	5	1.92
5	$((C' \cap A) \cup (B' \cup A)')$	100%	6	1.88
6	$((C \cap A) \cup (A \cup B)')'$	100%	5	2.01
7	$((A' \cup C)' \cup (A' \cap B))$	100%	6	1.97
8	$((B \cap A') \cup (A \cap C'))$	100%	5	2.08
9	$((A \cap C)' \cap (B \cup A))$	100%	4	1.97
10	$((C' \cup A') \cap (B \cup A))$	100%	5	2.45

Table 13: Full results for 3-input problem (using 'Do Not Care' in truth table)

Run #	Solution	Correctness	Size	Walltime (s)
1	$(B \cup A)$	75%	1	1.12
2	$(A' \cap B)$	75%	2	1.12
3	$(A' \cup C)'$	75%	3	1.11
4	$((C \cap A) \cup (A \cup B)')'$	100%	5	1.81
5	$((A \cup B')' \cup (C' \cap A))$	100%	6	1.72
6	$(A' \cup C)'$	75%	3	1.20
7	$((A \cup B')' \cup (A \cap C'))$	100%	6	1.97
8	$((B \cup A) \cap (C \cap A)')$	100%	4	1.76
9	$(A \cup B)$	75%	1	1.14
10	$(A' \cap B)$	75%	2	1.11

Table 14: Full results for 4-input problem (Most solutions too large to show)

Run #	Correctness	Size	Walltime (s)
1	81%	17	3.86
2	75%	7	3.95
3	94%	14	8.44
4	88%	10	6.93
5	81%	13	4.55
6	88%	19	7.47
7	69%	4	2.80
8	88%	14	6.32
9	94%	19	8.05
10	81%	14	4.26