DivSearch: Stochastic Optimization of Polynomial Arithmetic Circuits With Loose Constraints

Adam Klein
aklein4@stanford.edu

Sophie Andrews
sophie1@stanford.edu

Santino Ramos
santino@stanford.edu

ABSTRACT

In this paper, we analyse the performance of various combinatorial optimization methods for generating arithmetic circuits of polynomials, as well as demonstrate a new algorithm that yields significantly more efficient circuits. Polynomial circuits are important in several fields, such as cryptography and simulated physics. As the efficiency of each circuit can have a direct impacts on computational cost, there is significant incentive to improve their design. Prior work in circuit optimization only focuses on optimizing specific types of polynomials, or using expert-crafted rules to restrict the problem's search space and give locally optimal solutions. By treating the problem as a naive combinatorial optimization problem, we test global-search and stochastic sampling methods to find solutions that span the entire search space. Then, combining our combinatorial approach with the existing multivariate Horner's scheme, we propose DivSearch: a new algorithm that uses combinatorial optimization to operate over a less restricted solution space than the original scheme. By factoring polynomials using arbitrary pre-computed monomials, rather than single variables, we are able to create superior solutions than those possible with the original scheme. To find these solutions in the large search space, we apply the metaheuristics simulated annealing and basin-hopping, as well as a guiding heuristic. Our results show that across various sizes of polynomials, DivSearch is able to achieve up to a 20% improvement in circuit efficiency versus existing multivariate Horner's scheme implementations.

INTRODUCTION

The process of evaluating a polynomial can be represented with a directed acyclic graph (DAG) structure known as an arithmetic circuit. Leaf nodes represent constants and variables. Each parent node represents an operation on its child nodes, with the operations for our problem being confined to addition or multiplication. As we move from the leaf nodes to the top of the circuit, we develop a representation for our polynomial.

A given polynomial can be represented by many different equivalent circuits, and different representations can require different numbers of operations. Our goal is to find the circuit for a given polynomial that has the fewest operations, as that makes it the most efficient to evaluate. Specifically, we care most about the number of multiplications, as it is the more expensive operation.

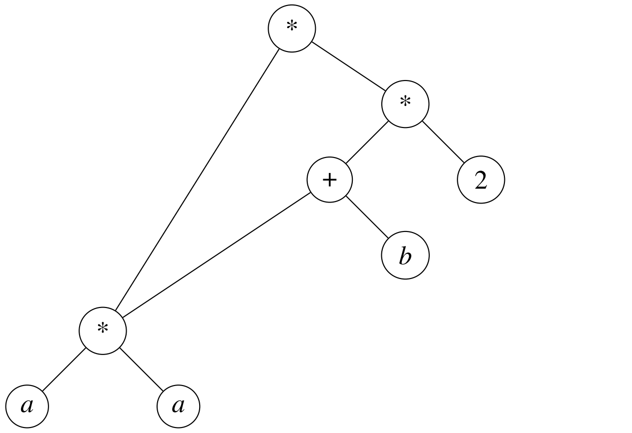


Figure 1: A circuit representation for $2a^4 + 2a^2b$: As we work our way up the circuit, we first generate $a \cdot a$, then $b + a \cdot a$, then $2(b + a \cdot a)$, and our final multiplication operation gives us $2a^2(b + a \cdot a) = 2a^4 + 2a^2b$.

The use of an optimized arithmetic circuit will result in faster evaluation of the corresponding polynomial the circuit represents. This is an important consequence for many fields in which the speed of numerical computation is a limiting factor to progress. In particular, both high energy physics and cryptography rely on the use of large polynomials to compute important characteristics about the systems they are studying.

In the space of high energy physics, particle dynamics can be roughly modeled by multivariate polynomials, and Monte Carlo integration of these expressions is a common way to measure the outputs of certain reactions[4]. Physicists might need to evaluate the formulas millions of times to obtain accurate results, so having a representation of the formula that is as "short" as possible can significantly improve the speed at which different particle interactions are evaluated.

Similarly, arithmetic circuits play a key role in Zero-Knowledge proof systems — a subfield of cryptography that has been gaining lots of traction in recent years due to its important applications to cryptocurrencies and secure blockchain protocols. In the implementation of these systems, one party (the prover) has to prove to another party (the verifier) that they know a certain piece of information without revealing the actual information itself [7]. In practice, these proofs can have many different mathematical representations (hash function, pairings on elliptic curves, etc), but what most modern ZK compilers do (Zokrates, Circom, CirC) is take the higher-level representation and compile it down to an arithmetic circuit [9].

The verification of the proof becomes the same as checking if an input is a satisfiable assignment on inputs to the arithmetic circuit, which depending on the size and complexity of the proof, can be the biggest bottleneck in the runtime of the whole protocol.

Generating a simple circuit for a given polynomial is easy: operations can be assigned in exactly the same way that the polynomial is written, substituting exponents for series of multiplications. However, much more efficient circuits usually exist, and finding them is a historically difficult problem. Prior work finds efficient circuits for specific types of target polynomials [1], or limits the search space to single-variable factoring, as is the case for the popular multivariate Horner's scheme [8]. While these techniques often yield good results, their confined search spaces mean that the methods likely to find better, though harder to find, solutions. In our implementation, we explored multiple paths for optimization and settled on DivSearch, which takes any arbitrary large polynomial and finds an optimized circuit. Our DivSearch algorithm uses a combination of Horner's method and stochastic superoptimizatsearch space

RELATED WORKS

Prior methods for generating efficient arithmetic circuits have relied largely on expert-crafted heuristics and rules to constrain the problem's search space. For instance, one optimization strategy focuses on polynomials that just require addition [1] in circuit generation. However, our approach handles large polynomials with both addition and multiplication operations. Another optimization strategy requires the input to be a sparse univariate polynomial [3], but our input polynomials are multivariate and we do not restrict their characteristics. Given arithmetic circuits are so powerful, certain methods have explored how to approximate them, verify them, or correct ones that have been generated with errors [2, 13, 11]. Our optimization does not require specifications for the input polynomial, and finds an exact circuit.

One of the simplest strategies for combinatorial optimization is a brute force search through every possible circuit. This method has been a major focus of superoptimization research, which is similar to arithmetic circuit generation and seeks to generate the most efficient program to execute a given function [6]. However, brute force search can also be costly from a time and memory standpoint. Therefore, our optimization will use random search strategies.

At the cost of guaranteed completeness, the Markov Chain Monte Carlo (MCMC) sampling method has been shown to produce comparable results to exhaustive search [6] while being able to handle a larger search space [12]. Furthermore, for multivariate equations, random search has already been shown to be "faster than exhaustive search in very generic settings" [5]. Given the parallels between this task and our multivariate polynomial problem, we should hope to see similar results from stochastic approaches. For both constrained and unconstrained examples, controlled random search mechanisms are successful for global optimization, with the added benefit that it does not always sacrifice computer storage [10].

Within the domain of polynomial circuit optimization, there has been some work done on stochastic optimization for multivariate polynomial circuits. However, this has been limited to optimizing the order of variables for multivariate Horner's scheme optimization through Monte Carlo tree search [4]. However, using Horner's scheme in this manner severely limits the search space. We improve upon their work by expanding the method with stochastic superoptimization to a more expressive version of the scheme.

Horner's scheme

We provide an overview of the existing work done with Horner's scheme for circuit optimization as we expand upon it.

For a univariate polynomial, Horner's scheme is guaranteed to provide the solution that minimizes the number of operations in the circuit. Horner's Factorization of a degree n polynomial for efficient evaluation is described by:

$$\begin{aligned} a(x) &= a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n \\ &= a_0 + x(a_1 + x(a_2 + x(\ldots + x a_n))) \end{aligned} \tag{1}$$

In this form, the polynomial can be evaluated with n additions and n multiplications. Horner's method can be generalized to the multivariate case. In this case, one chooses a variable, treats the other variables as constants, and applies the factorization in Equation 2. This process is applied to each variable.

As an example, consider the polynomial $a = y - 3x + 5xz + 2x^2yz - 3x^2y^2z + 5x^2y^2z^2$ and processing the variables in the order x, y, z. The result of applying multivariate Horner's scheme is

$$a = y + x(-3 + 5z + x(y(2z + y(z(-3 + 5z))))) \tag{2}$$

which can be evaluated with 5 additions and 8 multiplications [4].

IMPLEMENTATION

We start with brute force and stochastic search implementations that are successful on smaller polynomials. However, their limitations are quickly revealed for the large multivariate polynomials we are interested in, so we use a more sophisticated algorithm. Our final implementation uses superoptimization applied to Horner's scheme. We used the different superoptimization approaches of the greedy algorithm, a random algorithm, simulated annealing, and basin hopping.

Initial attempts

Our initial attempts were the implementation of the relatively simple brute force and stochastic search. Using a brute force search for a polynomial is restricted by space complexity, and a purely stochastic search is not guaranteed to find a solution.

Brute-Force Complete Search

In its most basic form, brute-force search involves generating every circuit within a limiting constraint - which in our case is a maximum cost. We can set the maximum cost based on a solution that we already know, such as one generated by Horner's scheme, and use that fact that any circuits with a greater cost than the known solution is not helpful.

In the brute force search, we generate all trees up to a certain depth, and then we check for equality between our target polynomials and our newly generated circuits. We start with depth 0 trees, which are single nodes each containing a variable or a constant. A depth 1 tree is an operation applied to pairs of depth 0 trees. In a depth 2 tree, at least one child of the root is a depth 1 tree. We randomly select the other, and we continue using recursive definition to generate circuits with greater depth. Once we generate all our circuits, we search through them and check to see if one is equal to our target polynomial.

However, we quickly run out of space to store all circuits larger than depth 3, meaning that brute force search only works for polynomials that can be represented by a depth 3 circuit or smaller.

Stochastic search

For the stochastic implementation, we specify our target polynomial, and a set of constants for our circuits to use as building blocks (often, this is all non-zero integers of magnitude less than or equal to the largest coefficient in our polynomial). We also set a cost for both the addition and multiplication operations, such that the total cost of the circuit is the sum of all multiplications and additions in the circuit, weighted by their respective cost. While we care primarily about multiplications, setting the cost of addition to zero often leads circuits to blow up; therefore, we use a cost of 0.25 for addition and 1 for multiplication.

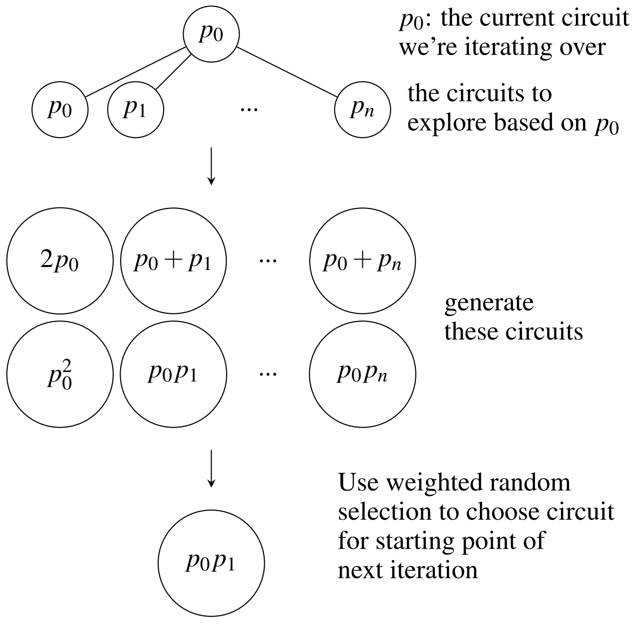

p_0
p_0: the current circuit we're iterating over
p_0
p_1
...
p_n
the circuits to explore based on p_0
$2p_0$
$p_0 + p_1$
...
$p_0 + p_n$
generate these circuits
p_0^2
$p_0 p_1$
...
$p_0 p_n$
Use weighted random selection to choose circuit for starting point of next iteration
$p_0 p_1$

Figure 2: The iterative stochastic search

To execute the search, we start with just the variables in our polynomials and the constants, and then add and multiply the pairs. For example, we may start with $a, b, c, 1, 2, 3$. Then we generate new circuits such as $a^2, 2a, ab, a+b, ac, a+c, a, a+1$ etc. We randomly select which circuit to use as the starting point in our next iteration based on a heuristic that takes into account the cost of the circuit and how far away it is from the cost of the target polynomial. In Figure 2, we demonstrate how we iteratively repeat this process, adding new operation gates to the base of the circuit and creating a larger and larger polynomial. If the cost gets too big, we effectively reset the search so the polynomials we are testing do not stray too far from our target.

For any iteration, we are only storing the current circuit we are exploring and the list of new circuits (e.g. [p_0, p_1 ... p_n] in Figure 2) we generate by applying operations. One new circuit to explore comes from this list. Therefore, we are storing significantly less information than in the brute force case, so we can search for much larger polynomials. However, the drawback to a stochastic search, given its random nature, is that we are not guaranteed to find a matching circuit for our target polynomial, even after thousands of iterations.

DivSearch

We found that our previous methods had trouble finding viable circuits, much less efficient ones. Even when using search methods strongly guided by heuristics, the circuits generated by ground-up methods often slightly missed the target polynomial. Therefore, we devised DivSearch as the response to this problem. Much like the existing Horner's scheme, DivSearch begins with the target polynomial and factors it down into single variables and constants - rather than trying to build the polynomial up from the base components. This means that every run of the algorithm produces a viable result, and much less time is wasted on trials that produced inaccurate polynomials.

The problem with multivariate Horner's scheme as a superoptimization tool is that it greatly constrains the search space. For a polynomial with n variables, Horner's scheme can produce $n!$ possible solutions. While the solutions produced by multivariate Horner's scheme are often very good, the solution space of the algorithm is significantly reduced compared to the problem's original solution space, and the globally optimal solution is likely outside of that subspace. In order to find solutions that are closer to the global optimum, DivSearch loosens the constraints of Horner's scheme by factoring by arbitrary monomials, rather than only first-order single-variable monomials. Furthermore, rather than decided on the order of variables at the beginning of the process, DivSearch independently chooses a new monomial factor at each factorization step. This increases the decision space from a single permutation of the variables, to a lager number of independent choices. Therefore, DivSearch has a solution space of size $|M|^c$, where c represents the number of multiplications in the circuit (this is an approximation, as different solutions have different solutions have different numbers of multiplications), and M is the set of monomials that are being chosen from for factoring.

Algorithm 1 DivSearch

```
1: procedure DIVSEARCH(P)
2:     $D$ = set of randomly generated monomials
3:     for $x_i \in P$ do
4:         $D \leftarrow x_i$
5:     $D_{used}$ = set
6:     $G$ = stack
7:     $G \leftarrow P$
8:     while not empty($G$) do
9:         $Q \leftarrow G$
10:        remove 0th-order monomials from $Q$
11:        if $Q$ has monomials then:
12:            $d \leftarrow$ chosen monomial from $D$
13:            $D_{used} \leftarrow d$
14:            $A \leftarrow$ monomials in Q divisable by d
15:            $B \leftarrow Q - A$
16:            $G \leftarrow B$
17:            $A = A/d$
18:            $G \leftarrow A$
19:    for $d_{used} \in D_{used}$ do
20:        $f \leftarrow d_{used}$
21:        while $f \neq x \wedge (f \notin D_{used} \vee f = d_{used})$ do
22:            $f = f/($ random $x \in f)$
```

The maximum reasonable monomial set for an n variable polynomial of maximum degree r is:

$$M = \left\{ \prod_{i=1}^{n} x_i^{k_i} \,\middle|\, \sum_{i=1}^{n} k_i > 0 \wedge \forall k_i \in \mathbf{W} <= r \right\} \tag{3}$$

However, large M can have a detrimental effect on performance when using a complicated heuristic. Therefore, smaller monomial sets are often chosen randomly in practice.

For an example of how much larger DivSearch's solution space is compared to multivariate Horner's scheme, consider an arbitrary fifth-order polynomial of five variables. Multivariate Horner's scheme can produce $5! = 120$ possible solutions for this variable. Assuming $c \approx 25$, the DivSearch search space with maximal M for this example has cardinality $(6^5 - 1)^{25} \approx 1.85 \times 10^{97}$, which is clearly many orders of magnitude larger than seen in multivariate Horner's scheme.

Since the search space of DivSearch is so much larger, it is likely to contain many possible solutions that are very inefficient. Therefore, we have deveoped two methods for finding good solutions: heuristics and stochastic optimization.

Heuristic

After much experimentation, we have found a heuristic for choosing monomial factors in DivSearch that produces relatively good results across many problems, while still giving good performance. In order to simlify notation and allow for vectorization, we use the following definition:

A monomial m that is part of a polynomial with n variables is described by a vector v in $\mathbf{W^n}$, such that v_i equals the exponent of x_i in m.

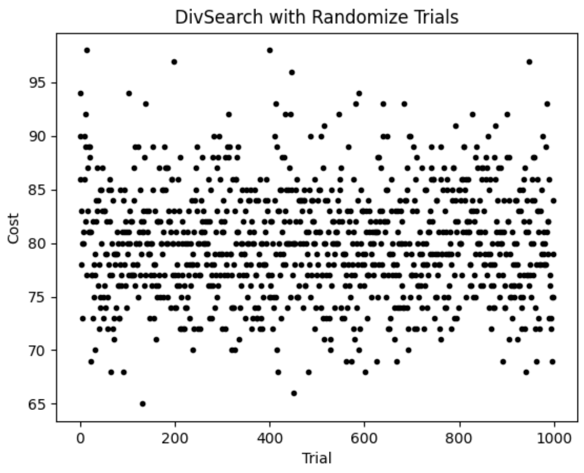


Figure 3: Random sampling progression with $\gamma = 3$ on a polynomial with 3 variables, 25 coefficients, and exponenets sampled from an exponential distribution of scale 3.

Using this definition, our heuristic is calculated by assigning a score to each factoring monomial d in the set of choices M using equation, and taking the monomial that yields the highest score.

$$score(d) = \sum_{m \in P} \begin{cases} 0 \; if \; \min(m-d) < 0 \\ \lim\limits_{\varepsilon \to 0}((d + \varepsilon \mathbf{1}^n)^T (m + \varepsilon \mathbf{1}^n)^{\circ -1/2}) & else \end{cases} \tag{4}$$

Stochastic Search

In addition to making greedy decisions based on the above heuristic, we used three stochastic decision processes to generate many viable solutions, and take the best one: random sampling, simulated annealing, and basin hopping.

Random sampling simply chooses monomials probabilistically, rather than deterministically, and generates many unique samples. Monomials are chosen with relative probabilities weighted by their score raised to some power γ:

$$P(D = d) = score(d)^{\gamma} \tag{5}$$

When γ is zero, all monomials have the same probabity, making all solutions equally likely. When γ goes to infinity, it becomes greedy search. We have found that γ values in the range of $[1, 10]$ seem to work best, with lower γ values being more expressive and giving better results for smaller polynomials.

To improve upon random sampling, we applied the well-known simulated annealing meta-heuristic to the factorization circuits. Beginning with an initial solution generated by greedy or random sampling, we run many steps, at each step making a small change to the solution and evaluating the resulting cost. Changes are defined as replacing a single factoring monomial in the solution with a new one as described in random sampling, and replacing all subsequently affected factoring steps similarly.

Whether to accept the change and have it persist to the next step is decided probabilistically according to equation 3, where c_{old} is the cost of the current solution before the change, c_{new} is the cost of the solution after the change, and T is the temperature, which decreases to near zero as the program proceeds.

$$P(accept) = \begin{cases} 1 & c_{new} <= c_{old} \\ exp\left(\frac{c_{old}-c_{new}}{T}\right) & else \end{cases} \tag{6}$$

When T reaches near zero, the algorithm becomes hill-descent, only accepting changes if $c_{new} <= c_{old}$.

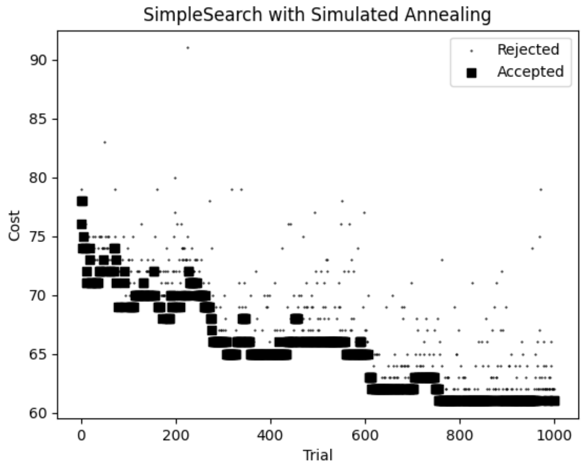


Figure 4: Simulated annealing progression with $\gamma = 3$ on a polynomial with 3 variables, 25 coefficients, and exponents sampled from an exponential distribution of scale 3.

Basin-hopping is an extension of the final stages of simulated annealing, where changes are only accepted if they do not make the solution worse. This is done for many initial solutions generated using random sampling.

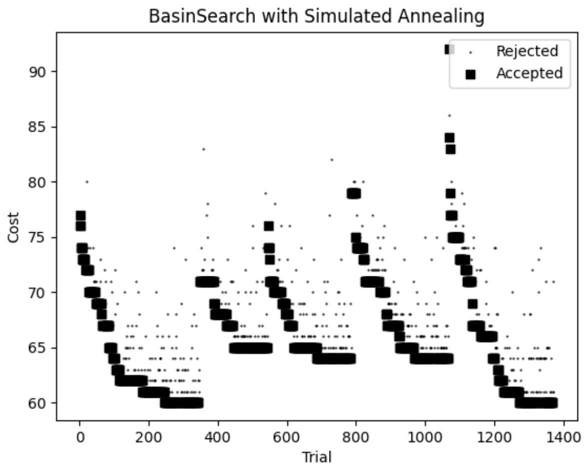


Figure 5: Basin-hopping progression with $\gamma = 3$ on a polynomial with 3 variables, 25 coefficients, and exponenets sampled from an exponential distribution of scale 3.

EVALUATION

As stated previously, we care most about minimizing the number of multiplication operations in a given circuit. Therefore, this is our primary metric of evaluation.

For a comparison, we use the open-source multivariate Horner's scheme library *multivar_horner* [4]. We used the *HornerMultivarPolynomialOpt* function from the library, which uses A* search to produce the library's best results.

For testing, we generated 100 examples of polynomials of various sizes, and compared the best resulting circuits from each of *multivar_horner*, greedy DivSearch, random sampling DivSearch, simulated annealing DivSearch, and basin-hopping DivSearch.

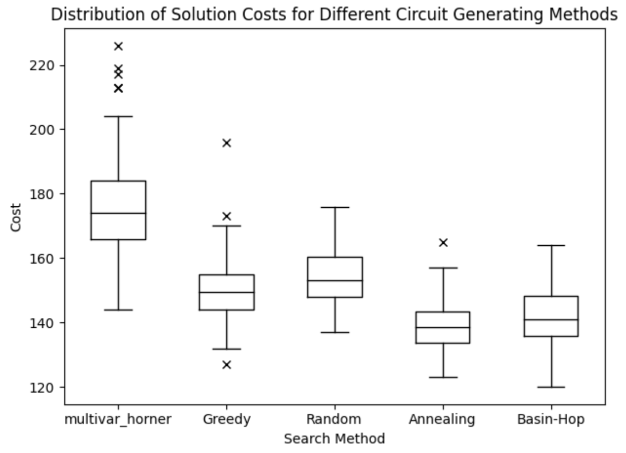


Figure 6: Comparison of algorithms across 100 trials of polynomials with 5 variables and 100 coefficients, with exponents sampled from an exponential distribution of scale 2.

For the particular set of examples used in Figure 6, each polynomial had 5 variables and 25 coefficients, and the expenent of each variable multiplied by each coefficient was samled from an exponential distribution of scale 2. We see that DivSearch significantly outperformed *multivar_horner* across all of its optimization techniques.

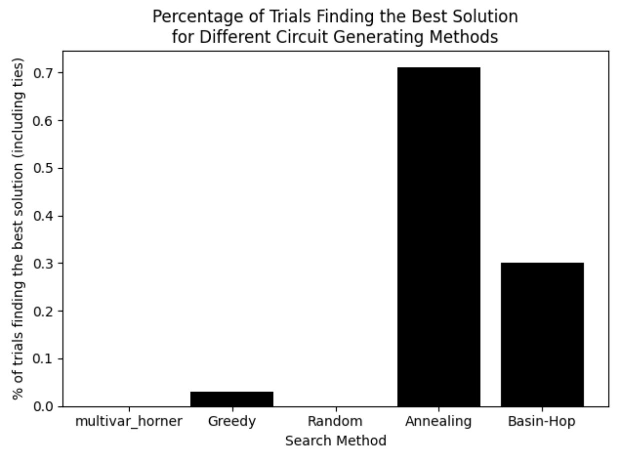


Figure 7: The frequency with which each algorithm produces the best solution, across 100 trials of polynomials with 5 variables and 100 coefficients, with exponents sampled from an exponential distribution of scale 2.

Here we see the rate at which each algorithm produced the best solution for each example. Simulated annealing produced by far the most 'winning' solutions, followed by basin-hopping and greedy search. Interestingly, we note that in none of the 100 examples did *multvar_horner* or random sampling produce the best solution. However, as seen in the full performance chart in Appendix A, this is not the case across all tests.

The common trend throughout testing was that DivSearch tended to do better on polynomials with larger exponents. Furthermore, smaller polynomials allowed for only small improvements, if any. For small polynomials, random sampling proved the best, while more greedy approaches did better on larger polynomials. The benchmark library only outperformed DivSearch on one example, where the ratio of variables to exponents and coefficients was very high. This makes sense, as these types of polynomials would benefit less from being factored by larger monomials.

For a specific example of how DivSearch's solutions compare to *multivar_horner*, see Appendix B.

CONCLUSION

intro to the conclusion

Discussion

Future Work

A challenge for our group during the evaluation steps of this project was finding a widely-used arithmetic circuit for which a baseline calculation compared to our optimized calculation would show a significant improvement. Given our choice of a graph representation for the circuits, we could not find circuits in "useful" applications that also encoded their circuits in this format. This prompted our group to explore Rank-1 Constraint Systems (R1CS), which is the circuit representation used in Zero-Knowledge Proofs that were briefly outlined in the motivation section. An R1CS basically "unrolls" our graph into a collection of 1-line constraints that have 1 operation each. The number of constraints in an R1CS roughly maps to the number of multiplication gates in arithmetic circuits (as some constraints are redundant and can be combined/removed), so performing an optimization on this series of constraints would effectively be the same problem as what we do for the circuit graphs. However, adapting our algorithm to work with the R1CS format,

ACKNOWLEDGEMENTS

We would like to thank Matthew Sotoudeh and Michael Bernstein for their mentorship throughout this project.

REFERENCES

[1] Anup Hosangadi, Farzan Fallah, and Ryan Kastner. 2006. Optimizing High Speed Arithmetic Circuits Using Three-Term Extraction. EDAA. https://past.date-conference.com/proceedings-archive/2006/DATE06/PDFFILES/11F_1.PDF

[2] Honglan Jiang, Cong Liu, Leibo Liu, Fabrizio Lombardi, and Jie Han. 2017. A Review, Classification, and Comparative Evaluation of Approximate Arithmetic Circuits. In *ACM Journal on Emerging Technologies in Computing Systems*. ACM, 1—34. https://dl.acm.org/doi/abs/10.1145/3094124

[3] Pascal Koiran. 2012. Arithmetic circuits: The chasm at depth four gets wider. In *Theoretical Computer Science*. 56—65. https://www.sciencedirect.com/science/article/pii/S0304397512003131

[4] Vermaseren Herik Kuipers, Plaat. 2013. Improving multivariate Horner schemes with Monte Carlo tree search. https://www.sciencedirect.com/science/article/pii/S0010465513001689

[5] Daniel Lokshtanov, Ramamohan Paturi, Suguru Tamaki, Ryan Williams, and Huacheng Yu. 2017. Beating Brute Force for Systems of Polynomial Equations over Finite Fields. In *Proceedings of the 2017 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. ACM-SIAM, 2190–2202. https://epubs.siam.org/doi/epdf/10.1137/1.9781611974782.143

[6] Henry Massalin. 1987. Superoptimizer – A Look at the Smallest Program. ACM, 122—126. https://dl.acm.org/doi/pdf/10.1145/36177.36194

[7] Hartwig Mayer. 2016. zk-SNARK explained: Basic Principles. *URL https://blog. coinfabrik. com/wp-content/uploads/2017/03/zkSNARK-explained_basic_principles. pdf* (2016).

[8] Michelfeit. 2020. multivar horner: a python package for computing Horner factorisations of multivariate polynomials. https://arxiv.org/pdf/2007.13152.pdf

[9] Alex Ozdemir, Fraser Brown, and Riad S. Wahby. 2020. CirC: Compiler infrastructure for proof systems, software verification, and more. Cryptology ePrint Archive, Paper 2020/1586. (2020). https://eprint.iacr.org/2020/1586 https://eprint.iacr.org/2020/1586.

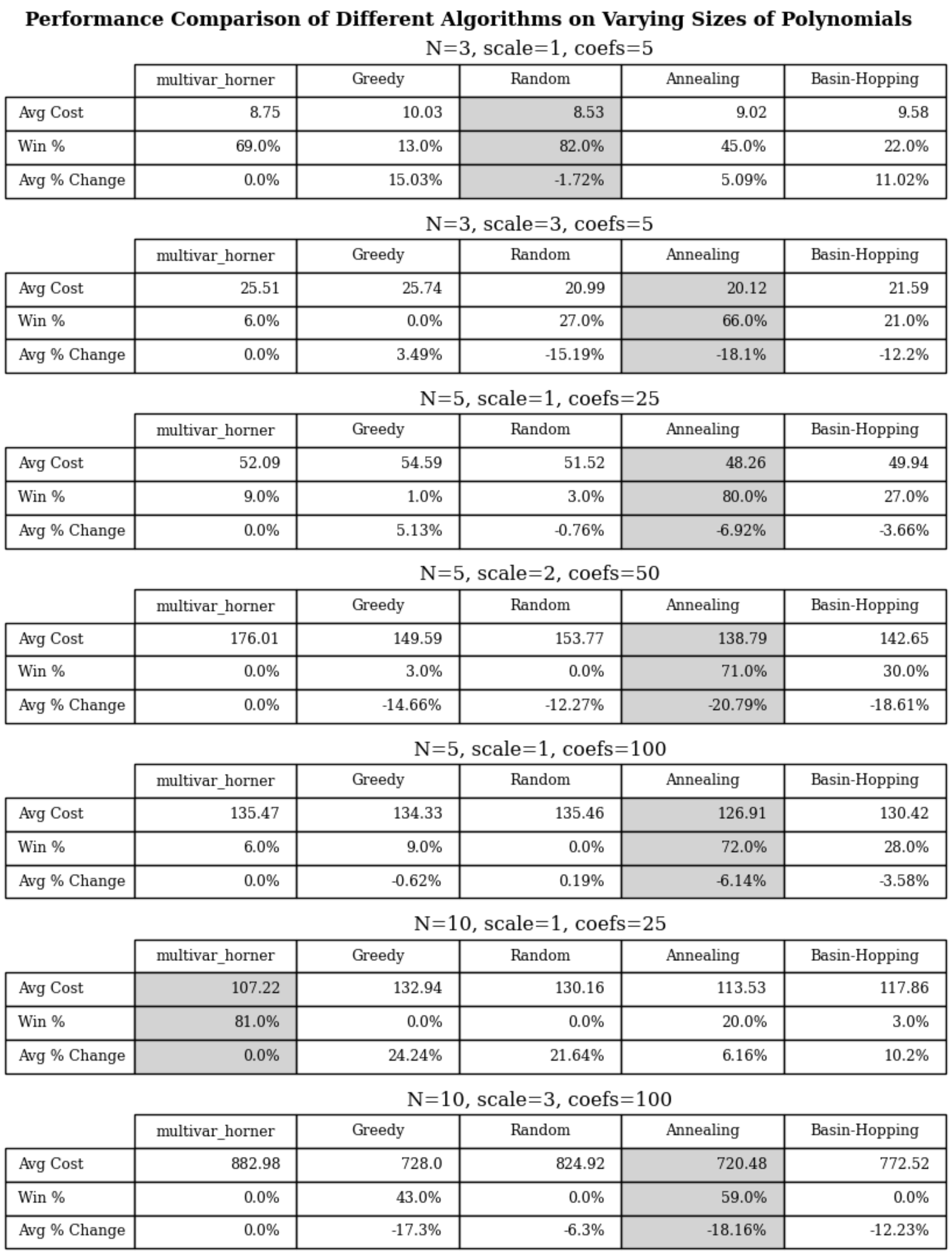
[10] W.L. Price. 1983. Global optimization by controlled random search. In *Journal of Optimization Theory and Applications*. 333—348. https://link.springer.com/article/10.1007/BF00933504

[11] Negar Sabbagh and Bijan Alizadeh. 2021. Arithmetic Circuit Correction by Adding Optimized Correctors Based on Groebner Basis Computation *(2021 IEEE European Test Symposium (ETS),)*. IEEE, 1—6. https://ieeexplore.ieee.org/document/9465454/authors#authors

[12] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic Superoptimization *(ASPLOS'13)*. https://theory.stanford.edu/~aiken/publications/papers/asplos13.pdf

[13] Yuki Watanabe, Naofumi Homma, Takafumi Aoki, and Tatsuo Higuchi. 2008. Arithmetic module generator with algorithm optimization capability. In *2008 IEEE International Symposium on Circuits and Systems*. IEEE, 1796–1799.

APPENDIX A.



Performance Comparison of Different Algorithms on Varying Sizes of Polynomials

N=3, scale=1, coefs=5

	multivar_horner	Greedy	Random	Annealing	Basin-Hopping
Avg Cost	8.75	10.03	8.53	9.02	9.58
Win %	69.0%	13.0%	82.0%	45.0%	22.0%
Avg % Change	0.0%	15.03%	-1.72%	5.09%	11.02%

N=3, scale=3, coefs=5

	multivar_horner	Greedy	Random	Annealing	Basin-Hopping
Avg Cost	25.51	25.74	20.99	20.12	21.59
Win %	6.0%	0.0%	27.0%	66.0%	21.0%
Avg % Change	0.0%	3.49%	-15.19%	-18.1%	-12.2%

N=5, scale=1, coefs=25

	multivar_horner	Greedy	Random	Annealing	Basin-Hopping
Avg Cost	52.09	54.59	51.52	48.26	49.94
Win %	9.0%	1.0%	3.0%	80.0%	27.0%
Avg % Change	0.0%	5.13%	-0.76%	-6.92%	-3.66%

N=5, scale=2, coefs=50

	multivar_horner	Greedy	Random	Annealing	Basin-Hopping
Avg Cost	176.01	149.59	153.77	138.79	142.65
Win %	0.0%	3.0%	0.0%	71.0%	30.0%
Avg % Change	0.0%	-14.66%	-12.27%	-20.79%	-18.61%

N=5, scale=1, coefs=100

	multivar_horner	Greedy	Random	Annealing	Basin-Hopping
Avg Cost	135.47	134.33	135.46	126.91	130.42
Win %	6.0%	9.0%	0.0%	72.0%	28.0%
Avg % Change	0.0%	-0.62%	0.19%	-6.14%	-3.58%

N=10, scale=1, coefs=25

	multivar_horner	Greedy	Random	Annealing	Basin-Hopping
Avg Cost	107.22	132.94	130.16	113.53	117.86
Win %	81.0%	0.0%	0.0%	20.0%	3.0%
Avg % Change	0.0%	24.24%	21.64%	6.16%	10.2%

N=10, scale=3, coefs=100

	multivar_horner	Greedy	Random	Annealing	Basin-Hopping
Avg Cost	882.98	728.0	824.92	720.48	772.52
Win %	0.0%	43.0%	0.0%	59.0%	0.0%
Avg % Change	0.0%	-17.3%	-6.3%	-18.16%	-12.23%

Figure 8: Caption

APPENDIX B.

Let us consider the following example polynomial:

$$P = c_1x_1x_2 + c_2x_1^2x_2 + c_3x_1x_2x_3 + c_4x_1^6x_2^2 + c_5x_2$$

Here, c_i's represent arbitrary constants and each x_i represents a variable. Below are illustrations of the circuits generated by each of *multivar_horner* using the *HornerMultivarPolynomialOpt* function (left), and by DivSearch with simulated annealing (right).

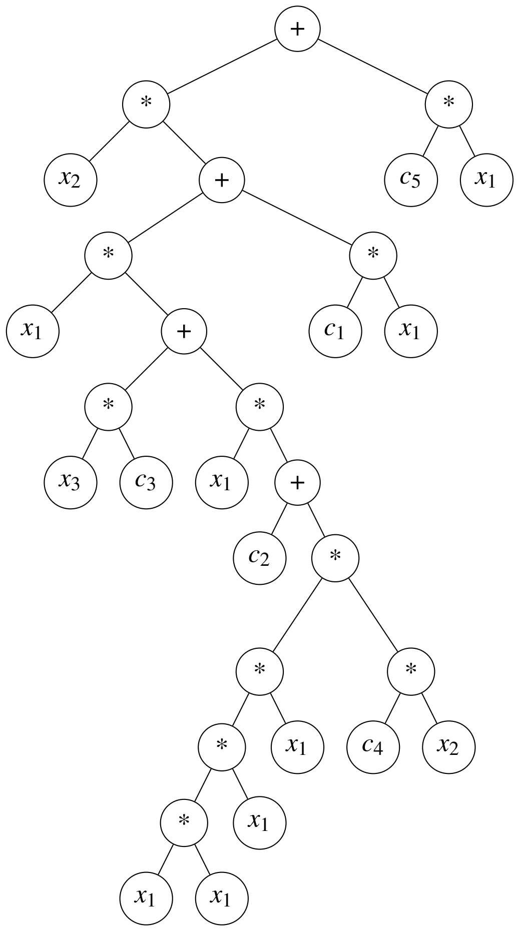


Figure 9: The circuit generated by the original Horner's scheme implementation for P

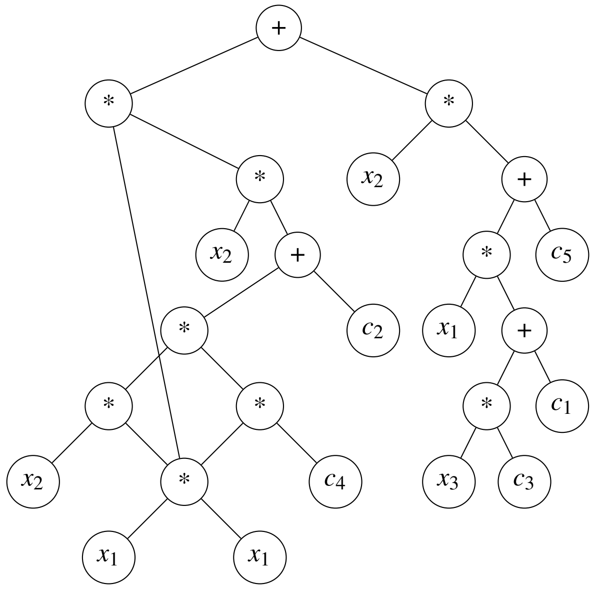


Figure 10: The optimized circuit generated by our DivSearch implementation for P

We see in Figure 10 our optimized circuit has 9 multiplications compared to 11 multiplications in Figure 9. Figure 10 also has fewer total operations: 13 compared against 15. This optimization is possible because in our implementation we allow a node to have more than one parent.