Optimization of Magnetic Tunneling Junction Devices for Neuromorphic Circuits for Solving MAXCUT

Ian Mulet[1,*], Bradley Theilman[2], Karan P. Patel[1], Jarad Arzate[3], Andrew Maicke[3], J. Darby Smith[2], James B. Aimone[2], Suma G. Cardwell[2], Jean Anne C. Incorvia[3], Catherine D. Schuman[1,*]
[1]*Department of EECS, University of Tennessee, Knoxville, TN, USA*
[2]*Sandia National Laboratories, Albuquerque, NM, USA*
[3]*Department of ECE, University of Texas at Austin, Austin, TX, USA*
[*]Corresponding Author Emails: imulet@vols.utk.edu, cschuman@utk.edu

***Abstract*—Novel algorithms leveraging neuromorphic computation are on the forefront of algorithm design. Here, we investigate how stochastic devices integrate and perform with a novel neuromorphic algorithm for solving MAXCUT problems in graphs. We evaluate how using magnetic tunneling junctions (MTJs) as the device to generate random numbers impacts the neuromorphic MAXCUT algorithm. We use both experimental MTJ data, as well as a model of the device behavior to investigate MTJ performance on this task. We also leverage the use of evolutionary optimization to tune the MTJ device to maximize performance on the algorithm and minimize energy usage of the device.**

I. Introduction

Novel algorithms leveraging neuromorphic computers offer an exciting path forward for improving computational and/or energy efficiency on many applications, including several different graph and optimization algorithms [1]. At the same time, a wide variety of emerging devices are being evaluated for neuromorphic computing implementations [2]. However, most of these neuromorphic device implementations are evaluated in the context of neural network applications (i.e., for training or inference of neural networks), not in the context of these new, emerging neuromorphic algorithm types. It is extremely important to consider these novel neuromorphic devices and novel neuromorphic applications together in order to gain a better understanding of how they perform collectively, and further, to understand which device types make the most sense for which applications. Moreover, there is an opportunity to tune device behaviors for given applications in order to get the best performance for a given application. We have previously shown that using novel devices, we can fine tune and optimize for specific applications and achieve better performance [3], [4]; however, to our knowledge, optimization approaches have not been applied to tuning neuromorphic devices for non-neural network applications.

One emerging device type of interest to neuromorphic computing and other emerging compute and memory more broadly is magnetic tunneling junction (MTJs). MTJs can be leveraged to produce true random number generation (TRNG) [5]. Exploration of TRNGs has been extensive in the security field as the true randomness of TRNGs enable them to be more cryptographically secure [6]–[9]. Similarly, MTJs have been extensively investigated as devices used in neuromorphic systems [10]. Furthermore, previous work has been done with integration of FPGA generated stochastic bits in regard to energy optimization [11]. However, in the neuromorphic field, there has been little testing done with integrating MTJs as TRNGs in non-neural network applications. In this work, we seek to close that gap by showing the performance of MTJ-based RNGs integrated with a novel neuromorphic circuit solving MAXCUT. We leverage both experimental MTJ data as well as a device model based on this data. Using the device model, we further use evolutionary algorithms to tune device characteristics to optimize for improved energy efficiency when executing the novel MAXCUT solver using these devices. We show that we are able to obtain improved energy efficiency with comparable algorithm performance to the default device parameters, indicating that there is an opportunity to optimize these devices for the MAXCUT application.

II. Related Work

Several attempts at improving traditional MAXCUT algorithm performance have been noted in previous literature [12]–[16]. However, there is relatively little literature on solving MAXCUT with neuromorphic circuits, and it is completely novel to integrate MTJ-based TRNG with these circuits.

TRNGs recently have been integrated into several applications, primarily focused on security. TRNGs offer the char-

We acknowledge support from the DOE Office of Science (ASCR / BES) Microelectronics Co-Design project COINFLIPS. We thank Lindsey Aimone for copyediting assistance.

979-8-3315-4127-9/24/$31.00 ©2024 IEEE

acteristic of being derived from physical-system interaction rather than traditional PRNGs relying on a seeded value. This is not the first MTJ-based TRNG model as Amirany et al. have designed and tested a similar TRNG utilizing MTJs and carbon nanotube field-effect transistors (CNTFET). This TRNG was proven to show true randomness via NIST testing, and when compared to previous TRNGs had much lower power consumption and energy expended per bit. This design shows how MTJ-based TRNGs have the lead edge over other TRNGs when it comes to low power design [17].

Demonstrating feasibility and performance has been long tried in the realm of TRNGs and security. Several applications have been integrated with TRNGs to demonstrate the potential for stronger security measures in Internet of Things (IoT) devices. Extensive experiments to show how applications utilizing TRNGs are cryptographically safe while also accounting for true randomness [18].

In our previous work, we have shown results of reinforcement learning and evolutionary algorithms when tuning for energy performance in a codesign workflow. This was done by using the MTJ device model to generate a gamma distribution [3], [4]. In this work, we expand upon that by replacing the simple application of generating a gamma distribution with a novel neuromorphic solution to the MAXCUT problem.

III. Methods

A. MAXCUT

The MAXCUT algorithm is an NP-complete problem that takes a graph's vertices and separates them into two different disjoint sets such that the separation maximizes the number of edges spanning between the two sets. Several different methods have been created to solve the MAXCUT problem. One such algorithm is the Goemans–Williamson algorithm, which features a semi-definite programming algorithm for solving an adjacency matrix from the graph [19]. Another algorithm that is utilized is the Trevisan algorithm. The Trevisan algorithm gets the cut weights by solving for the minimum eigenvector of the normalized adjacency matrix [20]. The Trevisan algorithm performs worse than the Goemans–Williamson algorithm in theory; however, in practice, it has a similar performance to the Goemans–Williamson algorithm [16].

Theilman et al. have devised two unique modifications to these algorithms to introduce neuromorphic circuit-based devices. This is done using popular neuromorphic components, including leaky-integrate and fire (LIF) neurons and Oja's anti-Hebbian plasticity rule [16], [21].

The LIF Goemans–Williamson algorithm has a variable we will refer to as dimensional matrix *W*, and the dimensions are defined by *n* (number of vertices) by *r* (rank of the solution). The neuromorphic circuit is defined as utilizing a number **r** random devices connected to *n* LIF neurons. A ratio is formed between the devices and LIF neurons which is mapped to the elements in the dimensional matrix *W*, giving the LIF covariances [16]. As Theilman et al. discussed, this enables the algorithm to take hardware specifications into account, by imposing a constraint on the range of weights. Finally, the cut weight is obtained by checking which neurons are spiking and not spiking and using that to assign the vertices in the corresponding disjoint sets [16].

The LIF Trevisan algorithm, while fundamentally a different algorithm, uses the same setup with an array of LIF neurons equaling the number of vertices in the graph associated with random devices. Specifically, the weight vector is obtained from funneling the population of LIF neurons into a single LIF neuron, and this is used to directly perform the Oja's anti-Hebbian plasticity rule [16], [22]. This results in the Trevisan algorithm's main objective of obtaining the minimum eigenvector by having the weight vector converge from the LIF covariance matrix [16].

The other two algorithms that we include in our evaluations are a standard Goemans–Williamson algorithm used for reference and a naive algorithm that randomly selects vertices for the cuts. These are used to compare performance against the LIF versions of the algorithms. As such we shall maintain that the standard Goemans–Williamson algorithm will be referred to as the *Solver*, modified algorithms will have LIF appended to them (LIF-GW, LIF-TR), and the naive algorithm shall be referred to as *Random* [16].

B. MTJ Device

The MTJ is a three-terminal, layered magnetic device comprising of a bottom ferromagnetic free layer (FL) separated from a top pinned layer (PL) by an insulating layer as shown in Fig. 1a. For devices built with perpendicular magnetic anisotropy (PMA), the magnetizations of these two layers will favor resting in the $\pm\hat{z}$ directions. When the magnetizations of the two magnetic layers are parallel the device is in a low-resistance state when reading across terminals T1 and T2, and when they are anti-parallel the device is in a high-resistance state.

While there are several ways to operate an MTJ device for TRNG, such as via voltage-controlled magnetic anisotropy or via spin-torque transfer, for the applications in this paper we consider MTJ devices based on spin-orbit torques (SOT). These SOT-MTJ devices rely on a heavy metal layer beneath the MTJ stack to host a spin-polarized current density J_{SOT} between terminals T1 and T3 to aid in the stochastic switching the magnetization of the FL $\hat{m}$. For an SOT-MTJ with PMA, the current through the heavy metal drives the FL magnetization into an unstable equilibrium in the plane of the device. Stochastic switching is then realized via the thermally driven symmetry breaking upon removal of the current, causing the FL to stochastically relax into either the $+$ or $-\hat{z}$ directions as shown in Fig. 1b. The binary interpretation of the resulting high- or low- resistance state as a '1' or a '0' thus gives a candidate binary random number generator. The pulse-and-relax scheme outlined here is capable of generating each bit on the order of tens of nanoseconds.

To study the potential applications of this device, we use a macrospin model outlined in [5] which considers the magnetization dynamics described by a modified Landau-Lifshitz-Gilbert equation. The effects of the heavy metal layer are

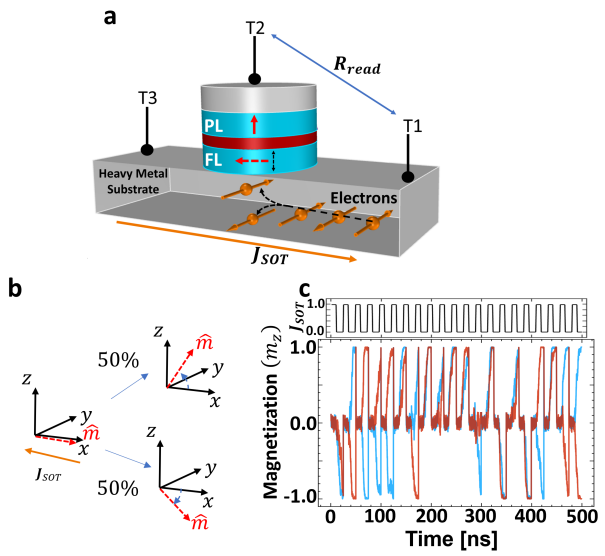


Fig. 1: Spin-Orbit Torque Magnetic Tunnel Junction (SOT-MTJ) device model and operation. a) Two magnetic layers, the pinned layer (PL) and free layer (FL) are separated by a thin insulating layer and sit atop a heavy metal substrate. When the magnetizations of the two layers are parallel, the device resistance R_{read} between terminals T1 and T2 is low, and when they are anti-parallel it is high, representing two distinct binary states. b) While the magnetizations of the two magnetic layers will rest out of plane for devices built with Perpendicular Magnetic Anisotropy, the magnetization of the free layer can be brought in-plane via application of a spin-polarized electric current J_{SOT} through the heavy metal between terminals T1 and T3. Removal of the current then causes stochastic relaxation into either the high- or low- resistance states with equal probability. c) Example bitstreams generated by an SOT-MTJ device, with application and removal of J_{SOT} indicated above the bitstreams.

described by the layer resistance, the spin-hall angle, and the spin polarization of the current. The other parameters, such as free layer anisotropy energy or saturation magnetization, are dependent on the device stack and are listed in table II. This paper will focus on the generation of uniform bit streams, as shown in Fig. 1c, and thus we utilize the various knobs available in the model to optimize the device for this application.

C. Test Graphs

During the execution of the integrated MAXCUT algorithm, several different graphs were used for testing purposes. First, for smaller graph sizes (50 - 500), graphs were randomly generated using the networkx library. The graphs were generated using `fast_gnp_random_graph` with each of the graphs being undirected. For the larger graphs, we used graphs based on real life examples from the Stanford Large Network Dataset Collection, which included a multitude of generated graphs from road networks to peer-to-peer network graphs [23].

We focused on selecting a graph that would be of adequate size, but could be run locally. We used a communication network graph, specifically the email communication network from Enron. The number of nodes in the graph is 36,692 and the number of edges is 183,831. With much larger graphs we ran into several memory-related issues when simulating the neuromorphic implementation. We could not run the algorithm on these graphs due to the graph sizes, which would require 250GB or more of memory.

IV. Results

A. Tera-bitstream

1) Experimental Setup: For these series of tests, we utilized a bitstream with a trillion bits that was directly sampled from a physical device. This physical device is an MTJ device that is actuated using a field programmable gate array (FPGA). The bitstream was tested against the NIST Statistical Test Suite to verify the randomness, in which it succeeded with only one XOR operation. This bitstream is unique in the sense that not only is it verified as truly random, but the bitrate and setup costs are heavily reduced. This bitstream has all the qualities that we need to utilize for MAXCUT, fulfilling the need for a stochastic device [24].

The initial testing of the tera-bitstream with the MAXCUT algorithm required integration of reading random bits, captured in a file from the FPGA-actuated MTJ from a stream, into the application. This was integrated into the MAXCUT algorithms to replace the default RNG devices; in particular, anywhere a random bit would have been sampled, we instead draw from this bitstream. For our testing we maintained much of the original MAXCUT variables, including graph sizes `50, 100, 200, 350, 500` and connectivity probabilities of `.10, .25, .50, .75`. We utilized 2^{20} different graph cuts for each iteration of the algorithm.

We conducted both integration and stress tests using the tera-bitstream. The integration tests followed the same setup as in the original MAXCUT implementation that Theilman et al described, which looked at a variety of graph sizes and connection probabilities. In our integration tests, each of the graph sizes and connectivity probabilities were run ten times. Then, we conducted a stress test to see how well the approach scales. The stress test was run using a large, real-world graph, in our case the Enron Email graph. The same number of cuts were performed on the graph as stated before 2^{20} [16].

TABLE I: Number of bits used

n	LIF Goemans–Williamson	LIF Trevisan
50	4,194,304	52,428,800
100	4,194,304	104,857,600
200	4,194,304	209,715,200
350	4,194,304	367,001,600
500	4,194,304	524,288,000
36692	4,194,304	38,474,350,592

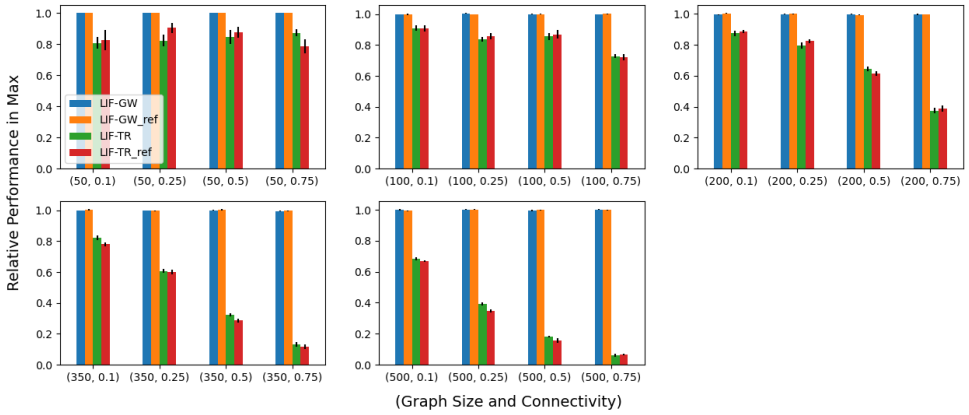


Fig. 2: Comparison of max cuts found for each of the graph and connectivity sizes with respect to the original Goemans–Williamson algorithm (solver). The n is the number of nodes and p is the connectivity probability. The comparison is between the MAXCUT algorithm that is powered by the trillion bitstream and a reference default MAXCUT run with normal PRNG's. The designation of *_ref* is given for the reference runs.

2) Results: In Table I, we can see the distribution of bits being used from the tera-bitstream. The unchanging number of bits required is an interesting quirk of the LIF Goemans–Williamson algorithm. The construction of the algorithm can be explained in two stages: solving for the SDP (semi-definite programming) and the sample stage. The solving the SDP stage is the same as the traditional Goemans–Williamson algorithm, but the interesting part is in the sampling section. The algorithm is set up so the rounding step is replaced by a sampling section. This section samples standard normal random variables, with one random variable per vertex. Theilman et al. postures that there is such a sequence of random variables that has an association to the unit vector in the solution. If that is the case, one can simply check the sign of the random variable to determine what the cut is assigned to. In regards to the LIF section, the sampling of standard normal random variables is instead used to generated a random population of LIF neurons. There is a set rank used within the function not only to shape the SDP solution vector, but to also have r number of devices be directly tied to LIF neurons. In our case, for testing, the rank is equal to four, which makes the number of bits consistent despite the graph size increasing [16].

Table I shows that the LIF Trevsian algorithms bit usage scales with the graph size. Oja's anti-Hebbian plasticity rule requires a pool of LIF neurons to be generated according to the size of the graph. This is done for every iteration that a cut weight is generated, so there is a direct relationship between graph size and the number of bits being utilized in the algorithm.

Figures 2 shows the comparison between the max cuts for each of the graph sizes with respect to the reference run of MAXCUT. Regarding the Goemans–Williamson algorithm comparison there is very little variation between reference LIF Goemans–Williamson and the bitstream-based LIF Goemans–Williamson algorithms. For the LIF Trevisan algorithm, we start to see variation between reference LIF Trevisan and the bitstream-based LIF Trevisan with the largest variation being displayed for the graph with 50 vertices. This behavior is expected, as with a smaller graph there will be higher variation and more apparent max cuts for that graph, as both the reference and bitstream-based Trevisan algorithm have a higher opportunity to converge the minimum eigenvector.

Figure 3 gives an overview of the performance of both algorithms with respect to the corresponding solver (reference Goemans–Williamson algorithm). As we can see the LIF-GW (leaky integrate and fire Goemans–Williamson) algorithm, with the tera-bitstream integrated, is performing exactly like the reference LIF-GW algorithm for each graph size and connectivity. For the LIF-TR (leaky integrate and fire Trevisan) and naive random cut selection algorithm, we start to see some variation for each of the graphs.

Looking at the tera-bitstream LIF Goemans–Williamson performance (blue line) we notice that the pattern from each of the graphs is that it closely matches the reference LIF Goemans–Williamson run (purple line is overlapping). This is very interesting since the sampling section of the algorithm directly relies on generating an appropriate sampling of random stochastic devices (in this case, sampling from a fair coin) to feed into the LIF neurons. As we can see, when using a bitstream generated from a physical device, the algorithm performs to the full capacity compared to the reference Goemans–Williamson algorithm. This behavior is exhibited on a wide range of graph sizes and connection probabilities, fully displaying the performance against a wide array of graph selections.

The performance of the tera-bitstream LIF Trevisan algorithm is also comparable to the reference when compared to the respective solvers. The sharp increase of performance for both the tera-bitstream and reference near the 10^3 to 10^6 can be observed on smaller graphs or graphs with a smaller

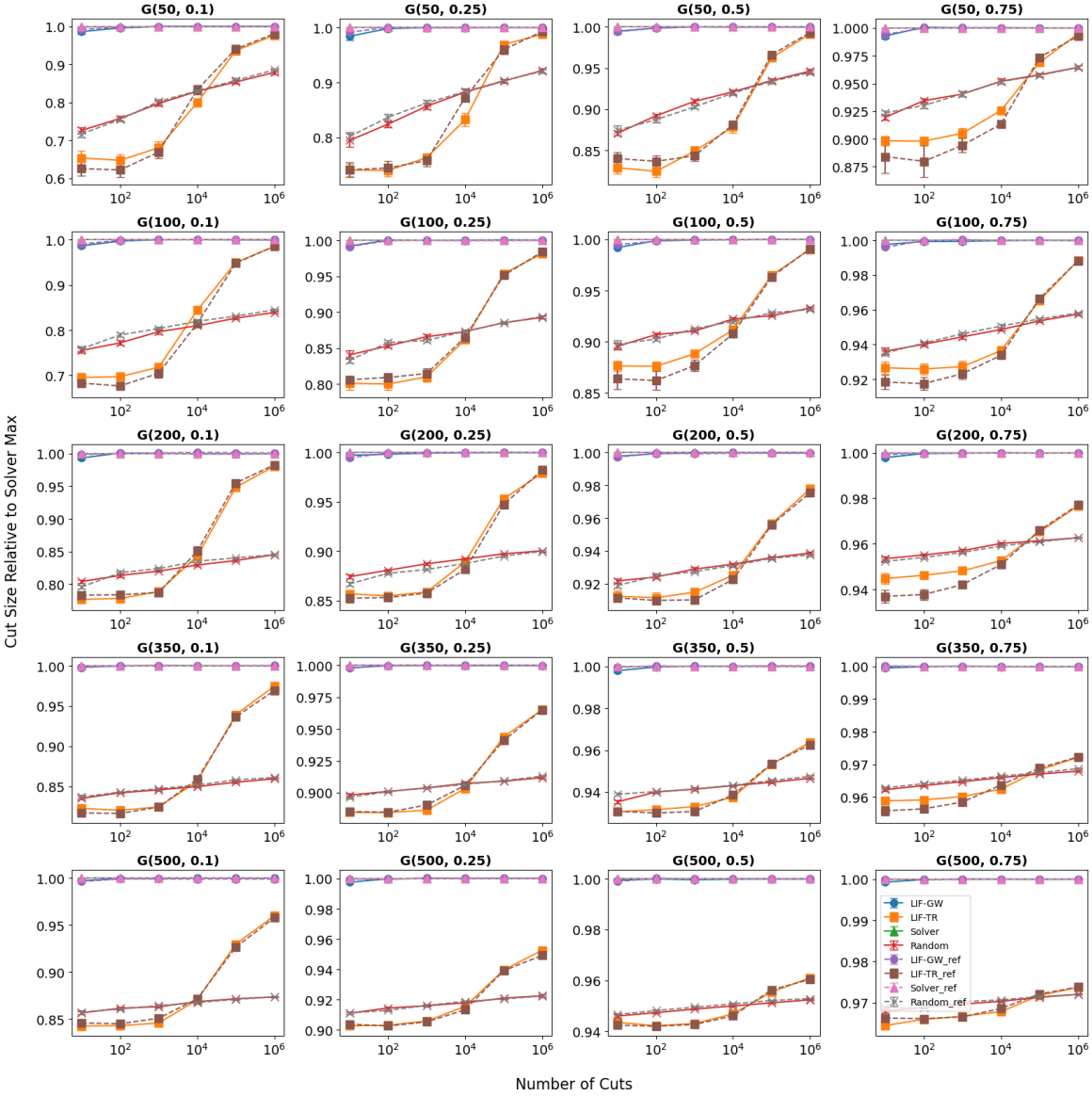


Fig. 3: Performance of the MAXCUT algorithm with integrated trillion bitstream. Each of the lines are maximum cut weight with respect to the Goemans–Willamson solver (labeled as 'solver' and the green and pink lines) as a function of the number of samples from Erdős–Rényi random graph. The rows are n number of vertices and the columns are p connectivity probability. The error bars are an average across 10 different graphs. There is overlap for LIF-GW (blue), LIF-GW_ref (purple), Solver (green), and Solver_ref (pink).

connectivity probability. Oja's anti-Hebbian plasticity rule is likely the cause for this, since this is an online learning method, the smaller graphs and less connected graphs have an easier time converging as the number of cuts increases. We postulate that the limiting factor for larger and more connected graphs is the limited number of cuts performed on the graph. Sadly, increasing the number of cuts quickly inflates the memory usage of the algorithm.

Interestingly, tera-bitstream LIF Trevisan performance (orange) in comparison shows an interesting variance when compared to the reference Trevisan performance (brown). The overlap between the tera-bitstream and reference is close enough that we can say the device achieved the purpose of being a stochastic device that produces TRNG behavior that will satisfy the needs of the Trevsian algorithm. Despite that, the performance is not entirely similar, and we can extrapolate that the randomness produced by the corresponding device (TRNG for the tera-bitstream and PRNG for reference) enabled the algorithm to converge more quickly. Specifically, in graphs 0.75 connectivity plots, the tera-bitstream performed better with convergence regarding a lower number of cuts. We can assume that the device that produced the tera-bitstream achieved a more true random distribution in accordance to the 50/50

uniform distribution. The relationship between the random devices and LIF neurons is directly tied to the convergence onto the minimum eigenvector, which generally speaking is the performance of the algorithm. That being said, if the random device, i.e. the tera-bitstream, can produce a higher quality random sample, it will enable the minimum eigenvector to converge quicker, increasing the cut size [16].

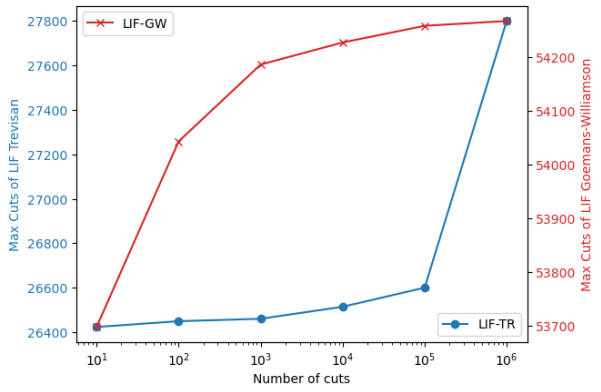


Fig. 4: Performance of the tera-bitstream and LIF Goemans–Williamson and LIF Trevisan algorithm when utilizing a large graph. The blue axes represents the LIF Trevisan algorithm and the red axes represents the LIF Goemans–Williamson algorithm.

Fig. 4 reinforces the shortcoming of the LIF Trevisan versus the LIF Goemans–Williamson algorithm. The clear difference in the max cuts at each interval is a clear indicator and when at the highest measured cut (10^6) we can see 51.2% lower max cuts. Interestingly, we see a large increase in max cuts between the 10^5 and the 10^6 cuts for the LIF Trevisan algorithm. This can be extrapolated from the previous graphs in Fig. 3 as the point where the performance of the LIF Trevisan algorithm starts to increase. It is safe to assume that we would see a very similar pattern to the graph sizes of 500 from Fig. 3 given there was a higher amount of cuts on the graph.

It is worth noting that memory utilization was an issue for this experiment as the LIF Trevisan, LIF Goemans–Williamson, and the reference Goemans–Williamson algorithm all have large matrix multiplications which quickly bloat the memory usage. It can be noted in Fig. 4, that there is no line for a reference Goemans–Williamson algorithm, and this is directly caused by a large matrix multiplication that cannot be avoided. We were able to find workarounds for both the LIF Trevisan and LIF Goemans–Williams algorithms. This figure still maintains validity after observing Fig. 3 and seeing the close performance of the LIF Goemans–Williamson compared to the reference Goemans–Williamson algorithm. It can be extrapolated that the performance of the LIF Goemans–Williamson with a large graph will maintain the same accuracy. Further steps can be taken given a machine with a large amount of memory, such as an HPC system. First, the reference Goemans–Williamson algorithm can be run to verify the LIF Goemans–Williamson algorithm. Second, a higher number of cuts can be applied to the graph which could be useful to see the performance of the LIF Trevisan algorithm. Lastly, larger graphs can be utilized to determine the performance of the scalability of the LIF Goemans–Williamson and LIF Trevisan algorithms.

B. Device Model

1) Experimental Setup: The introduction of the MTJ device model into the MAXCUT algorithm introduced the need for the integration of a new device for every random bitstream needed by the MAXCUT algorithm. Similar to the previous integration, the device model was also used to replace all locations where random bits are generated in the algorithm. That is, the model of the MTJ device was sampled for every bit needed by the MAXCUT algorithm. Specifically, the spin orbit torque (SOT) device model was utilized. We chose this model as it fit our needs regarding sampling a uniform distribution. Since the MAXCUT algorithm only needs a '0' or '1' state, we only want to focus on sampling from a uniform distribution. The spin transfer torque model (STT) was not chosen since the the SOT model offers a higher level of tuning [3]. It should be noted that when utilizing the model of the MTJ device this shifts the focus from the TRNG to PRNG (since the device model uses a PRNG as part of its modeling process). Another point to make is that the sampling of the device model will have a much worse throughput for bit generation. This is necessary for testing different device configurations as this enables a quick and easy testing suite for the MAXCUT algorithm. We can directly tune each of the parameters as needed while maintaining a clear-cut workflow.

TABLE II: MTJ model configuration

Parameter	Value
Alpha (Gilbert damping factor)	0.01
Ki (Anisotropy energy)	0.0002 J/m^2
Ms (Magnetic saturation)	300000 A/m
Rp (Resistance in the parallel state)	13265.5557Ω
TMR (Tunneling magneto resistance)	0.3Ω
eta (Spin hall angle)	0.8
J_she (SHE current density)	3.34994e+12 A/m^2
t_pulse (Pulse time)	7.5e-08 ns
t_relax (Relax time)	7.5e-08 ns
d (Thickness of the heavy metal layer)	3e-09 m
tf (Thickness of the free layer)	1.1e-09 m

Table II shows the configuration that we used for testing in the MAXCUT algorithm. These values were derived to ensure the device enters PMA state and allows the correct sampling of the uniform distribution.

2) Results: Fig. 5 follows the same pattern as Fig. 4 and this is to be expected. The device should be functioning similarly to the tera-bitstream, but within a software-based environment. It is important to emphasize that the device model is not a TRNG but a PRNG; here, we are validating the model's behavior against the real device, but our main goal is to leverage the model's ability to quickly tune and evaluate different device parameters. It should be noted that the LIF

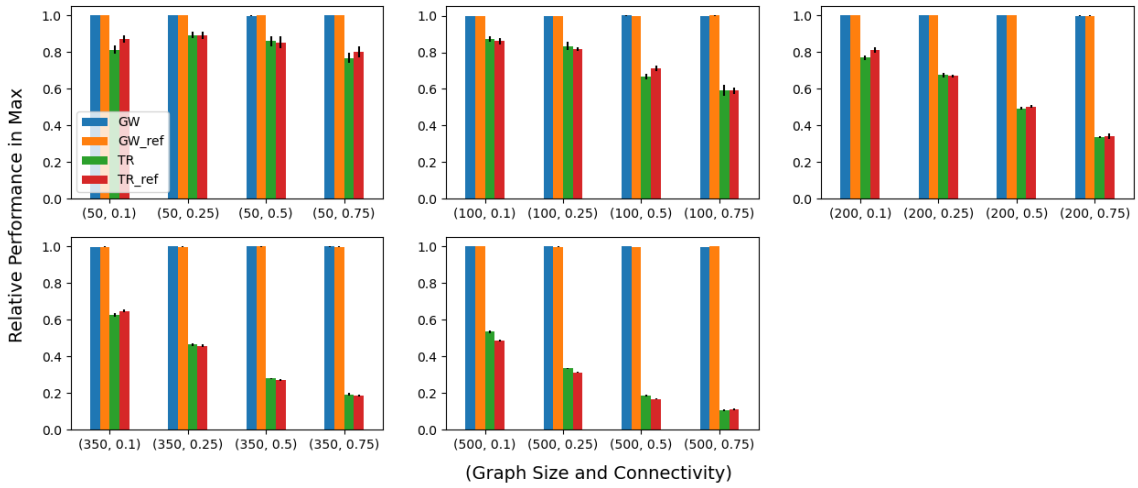


Fig. 5: Comparison of max cuts found for each of the graph and connectivity sizes with respect to the original Goemans–Williamson algorithm (solver). The *n* is the number of nodes and *p* is the connectivity probability. The comparison is between the MAXCUT algorithm that is powered by and a reference default MAXCUT run with normal PRNG's. The designation of *_ref* is given for the reference runs.

MAXCUT algorithms utilizing the device model are being compared against the reference MAXCUT run that is using the default PRNGs. Fig. 5 shows again that an MTJ device can fully function with these LIF MAXCUT algorithms while producing similar performance as the reference.

C. Evolutionary Optimization

1) Experimental Setup: Finally, we added the EA in order to optimize the device characteristics as referenced in II. Patel et al devised an experimental codesign workflow for both evolutionary and reinforcement learning algorithms for the MTJ device model [3]. An EA is one that is inspired by evolution and genetics. One of the most important components in an EA is the genome, which represents a solution to the problem that is being optimized. A set of these solutions are maintained throughout the algorithm as a population. A fitness function is used to determine how well the genome is performing. Several operators can be applied to each member of the population to produce offspring and possibly replace the parent if the fitness is better. In our scenario, we are optimizing for four different objectives: total energy, LIF Goemans–Williamson Algorithm performance, LIF Trevisan Algorithm performance, and Kullback-Leibler (KL) divergence from a uniform distribution. In particular, we are minimizing for the total energy and KL divergence, and maximizing for the algorithm performance. To implement the evolutionary algorithm, we are using LEAP-ec, which is a suite of evolutionary algorithms implemented in Python [25]. For this specific problem, we are using NSGA-II (non-dominated sorting genetic algorithm), which is a multi-objective optimization algorithm [26]. On the MAXCUT implementation side, we limited the graph sizes to 25 nodes and probabilities to .25. This was to ensure a reasonable running time for the optimization pass.

To calculate the performance metrics for the modified Goemans–Williamson and Trevisan algorithm, we calculated the difference under the curve with respect to the number of cuts and cut size relative to the naive random cut algorithm using the following equations, where *i* is starting index of the sum, *n* is the cut number, $h(x_i)$ is the difference between the two equations, and Δx_i is is the difference between the number of cuts:

$$R = \sum_{i=1}^{n-1} h(x_i) * \Delta x_i \tag{1}$$

$$h(x_i) = f(x_i) - g(x_i) \tag{2}$$

Where $f(x_i)$ is the LIF algorithm cut size at a relative cut amount (i.e. 10^2) and where $g(x_i)$ is the naive random cut algorithm cut size at a relative cut amount. Since the naive algorithm is the lowest-performing algorithm we should expect to see a larger number as the area between the curves (LIF and naive algorithm distance) increases over the entire graph. This justifies maximizing for this performance metric and ensures that the LIF algorithms are performing as expected.

A uniform distribution was sampled for the usage of this MTJ device as the MAXCUT algorithms are expecting a stream of `1's or 0's` and a KL divergence was calculated using the function:

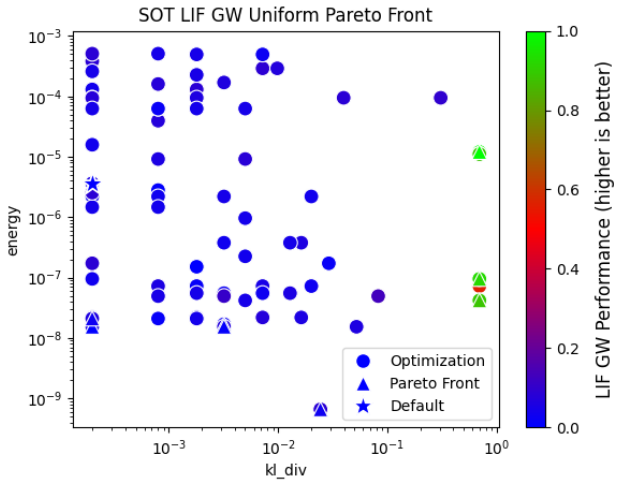
$$KL(P||Q) = \sum_x P(x) \log(\frac{P(x)}{Q(x)}) \tag{3}$$

Where P is the actual distribution sampled and Q is the theorized distribution. The KL divergence score was used to

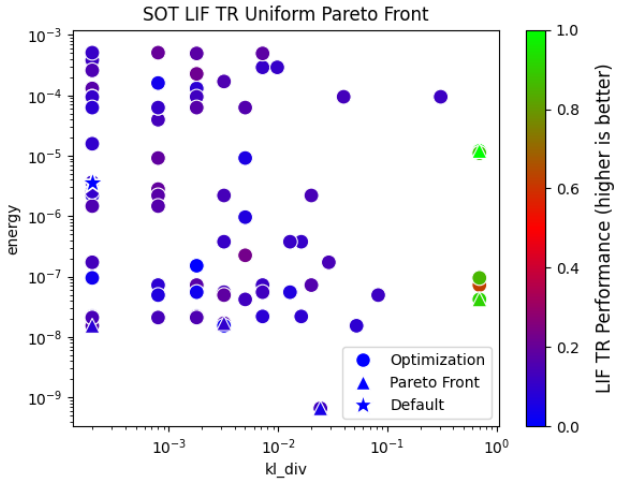
ensure that the uniform probability distribution was easily observable. A high KL divergence score would indicate a biased coin flip, and a lower score would indicate it is closer to a fair coin flip.

The range of the parameters that were modified during the optimization have been bounded to ensure that the model stays within the PMA state. This was done in accordance to the previous work [3]. The energy utilization is calculated as joules per bit generated.

The optimization framework was run with a population size of 30 for 30 generations. The evolutionary optimization tests were run using the specifications detailed above.



(a)



(b)

Fig. 6: Pareto fronts comparing energy and KL-Divergence. Each graph has a color mapping for how well the algorithm is performing, with each of the performance metrics separated into two graphs for each LIF algorithm. The star marker is to denote the default configuration, while the circular dots represent individual results from the optimization, and the triangles represent the Pareto front.

TABLE III: Default MTJ configuration vs. energy saving configuration

Parameter	Default	Optimization
Alpha (Gilbert damping factor)	0.01	0.01
Ki (Anisotropy energy)	0.0002 J/m^2	0.0002 J/m^2
Ms (Magnetic saturation)	300,000 A/m	1,825,292 A/m
Rp (Resistance in the parallel state)	13265.5557Ω	28604.3934Ω
TMR (Tunneling magneto resistance)	0.3Ω	0.3Ω
eta (Spin hall angle)	0.8	0.1
J_she (SHE current density)	3.34994e+12 A/m^2	2.8308e+11 A/m^2
t_pulse (Pulse time)	7.5e-08 ns	5.0e-10 ns
t_relax (Relax time)	7.5e-08 ns	5.0e-10 ns
d (Thickness of the heavy metal layer)	3e-09 m	3e-09 m
tf (Thickness of the free layer)	1.1e-09 m	1.1e-09 m
Total Energy	3.5743e-06 J	1.5314e-08 J

2) Results: In Fig. 6a, we can see the performance of the optimization with respect to the LIF Goemans–Williamson algorithm. There are several interesting things to note about the KL-divergence and the total energy usage. The color mapping on this graph is normalized against the default configuration run, and many of the genomes achieved equal or better results than that. When we look at the KL-divergence in relation to performance we can see a clear trend, as the KL-divergence approaches 1, the performance of the algorithm increases. This is a rather interesting result since this would mean that the distribution is behaving differently than the expected uniform distribution. If the distribution is skewed toward 1 or 0 this could drastically affect the performance of the LIF Goemans–Williamson algorithm. A likely scenario is the LIF algorithms are not performing better, but the naive random algorithm is performing worse. Since the performance metric is the area between the naive random algorithm and the LIF algorithm if the naive algorithm performs worse, than it will artificially inflate the performance metric. This is not ideal for accurately assessing performance and will be rethought in the future. When looking at the energy usage versus the performance there are clear groupings of higher performing metrics, but also higher than normal energy usage (specifically around the 13^{-3} – 10^{-4} range). Despite this, the Pareto front has several instances that use orders of magnitude less energy than the default configuration, and this shows that there is potential for vast improvement from the default configuration.

Why the data points share the same location as Fig. 6a is due to how the optimization was performed. The optimization is taking the total energy usage of both algorithms while the same KL-divergence is generated on a device configuration basis. In Fig. 6b one interesting thing to note about the performance is that many of the genomes resulted in better performance than the default configuration. We can see the cluster of data points at 10^{-2} tended to have higher performance than the default configuration. Furthermore, we have the same trend as from Fig. 6a as the KL-divergence approaches 1 the performance drastically increases. We can assume the reason is the same as the LIF Goemans–Williamson algorithm, the naive algorithm is performing much worse. The Pareto front data points were selected with the performance in mind, so there are slight variations between the LIF Goemans–Williamson and LIF

Trevisan algorithms, but we can see that many of the genomes selected were similar between the two.

In Fig. 6a and Fig. 6b, the majority of the genomes performed as well as the default configuration. Looking at KL-divergence and energy we see that if KL-divergence is slightly higher on average we can achieve lower energy, while maintaining the same performance. There is a clear cluster of data points around the 10^{-2} marker for KL-divergence which have a lower average energy than the default configuration. With respect to energy, all of the points on the Pareto fronts have achieved better energy utilization than the default configuration. Looking at both metrics while keeping both LIF algorithms performance in mind shows a clear picture: there are configurations that achieve the same performance, lowering energy, while the distribution is approximately close to fifty-fifty.

In Table III we can see the best performing configuration in regards to energy savings. Magnetic saturation, resistance in the parallel state, Spin Hall angle, SHE current density, and both pulse and relax time have been optimized. It show be noted that many of the top configurations that saved energy had similar Gilbert damping factor, anisotropy energy, tunneling magneto resistance, and thickness of both heavy metal and free layers. The total energy of the optimized configuration is 526% more efficient than the default configuration.

V. Discussion and Conclusion

There are a few key next steps that we intend to take from this research on novel MTJ devices and the novel neuromorphic MAXCUT algorithm. Many of the tests that we wished to run were not feasible due to hardware limitations. Many tests far exceeded our modest 256GB of memory on our local workstation, which limited our ability to explore how the Trevisan circuit behaves with higher cut sizes. With additional computation resources, the optimization runs also can be extended to cover more graph sizes and connectivities. A new performance metric needs to be derived for future optimization passes. This will only bolster the evidence of the performance of the novel MTJ device and the novel neuromorphic MAXCUT algorithm.

Evaluating and optimizing a novel MTJ-based TRNG for usage in a novel neuromorphic application is needed more than ever. We have demonstrated the feasibility of doing so, with a clear path for future work in scaling up the evaluations. We have shown several times that the performance in the MAXCUT algorithm utilizing the MTJ device rivals traditional PRNGs, offering a more tunable and energy-efficient method to generate random numbers. Leveraging these facts enables the MTJ device to be well-suited for neuromorphic algorithms that benefit from stochastic devices.

References

[1] J. B. Aimone, P. Date, G. A. Fonseca-Guerra, K. E. Hamilton, K. Henke, B. Kay, G. T. Kenyon, S. R. Kulkarni, S. M. Mniszewski, M. Parsa *et al.*, "A review of non-cognitive applications for neuromorphic computing," *Neuromorphic Computing and Engineering*, vol. 2, no. 3, p. 032003, 2022.

[2] J. Zhu, T. Zhang, Y. Yang, and R. Huang, "A comprehensive review on emerging artificial neuromorphic devices," *Applied Physics Reviews*, vol. 7, no. 1, 2020.

[3] K. P. Patel, A. Maicke, J. Arzate, J. Kwon, J. D. Smith, J. B. Aimone, J. A. C. Incorvia, S. G. Cardwell, and C. D. Schuman, "Ai-guided codesign framework for novel material and device design applied to mtj-based true random number generators," Submitted.

[4] S. G. Cardwell, K. Patel, C. D. Schuman, J. Darby Smith, J. Kwon, A. Maicke, J. Arzate, and J. A. C. Incorvia, "Device codesign using reinforcement learning," in *2024 IEEE International Symposium on Circuits and Systems (ISCAS)*, 2024, pp. 1–5.

[5] S. Liu, J. Kwon, P. W. Bessler, S. G. Cardwell, C. Schuman, J. D. Smith, J. B. Aimone, S. Misra, and J. A. C. Incorvia, "Random bitstream generation using voltage-controlled magnetic anisotropy and spin orbit torque magnetic tunnel junctions," *IEEE Journal on Exploratory Solid-State Computational Devices and Circuits*, vol. 8, no. 2, pp. 194–202, 2022.

[6] Z. Ji, J. Brown, and J. Zhang, "True random number generator (trng) for secure communications in the era of iot," in *2020 China Semiconductor Technology International Conference (CSTIC)*, 2020, pp. 1–5.

[7] R. Shiva Prasad, A. Siripagada, S. Selvaraj, and N. Mohankumar, *Random Seeding LFSR-Based TRNG for Hardware Security Applications*. Singapore: Springer Singapore, 2019, pp. 427–434.

[8] S. Taneja, V. K. Rajanna, and M. Alioto, "In-memory unified trng and multi-bit puf for ubiquitous hardware security," *IEEE Journal of Solid-State Circuits*, vol. 57, no. 1, pp. 153–166, 2022.

[9] V. Fischer, "A closer look at security in random number generators design," in *Constructive Side-Channel Analysis and Secure Design*, W. Schindler and S. A. Huss, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 167–182.

[10] J. Grollier, D. Querlioz, K. Camsari, K. Everschor-Sitte, S. Fukami, and M. D. Stiles, "Neuromorphic spintronics," *Nature electronics*, vol. 3, no. 7, pp. 360–370, 2020.

[11] W. A. Borders, A. Z. Pervaiz, S. Fukami, K. Y. Camsari, H. Ohno, and S. Datta, "Integer factorization using stochastic magnetic tunnel junctions," *Nature*, vol. 573, no. 7774, pp. 390–393, Sep 2019. [Online]. Available: https://doi.org/10.1038/s41586-019-1557-9

[12] D. Khilwani, V. Moghe, S. Lashkare, V. Saraswat, P. Kumbhare, M. Shojaei Baghini, S. Jandhyala, S. Subramoney, and U. Ganguly, "PrxCa1xMnO3 based stochastic neuron for Boltzmann machine to solve "maximum cut" problem," *APL Materials*, vol. 7, no. 9, p. 091112, 09 2019. [Online]. Available: https://doi.org/10.1063/1.5108694

[13] M. K. Bashar, A. Mallick, D. S. Truesdell, B. H. Calhoun, S. Joshi, and N. Shukla, "Experimental demonstration of a reconfigurable coupled oscillator platform to solve the max-cut problem," *IEEE Journal on Exploratory Solid-State Computational Devices and Circuits*, vol. 6, no. 2, pp. 116–121, 2020.

[14] M. M. Mohades and M. H. Kahaei, "An efficient riemannian gradient based algorithm for max-cut problems," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 69, no. 3, pp. 1882–1886, 2022.

[15] X. Zhao, Y. Li, J. Li, S. Wang, S. Wang, S. Qin, and F. Gao, "Near-term quantum algorithm for solving the maxcut problem with fewer quantum resources," *Physica A: Statistical Mechanics and its Applications*, vol. 648, p. 129951, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0378437124004606

[16] B. H. Theilman, Y. Wang, O. Parekh, W. Severa, J. D. Smith, and J. B. Aimone, "Stochastic neuromorphic circuits for solving maxcut," in *2023 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2023, pp. 779–787.

[17] A. Amirany, K. Jafari, and M. H. Moaiyeri, "True random number generator for reliable hardware security modules based on a neuromorphic variation-tolerant spintronic structure," *IEEE Transactions on Nanotechnology*, vol. 19, pp. 784–791, 2020.

[18] B. Lin, B. Gao, Y. Pang, P. Yao, D. Wu, H. He, J. Tang, H. Qian, and H. Wu, "A high-speed and high-reliability trng based on analog rram for iot security application," in *2019 IEEE International Electron Devices Meeting (IEDM)*, 2019, pp. 14.8.1–14.8.4.

[19] M. X. Goemans and D. P. Williamson, "Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming," *J. ACM*, vol. 42, no. 6, p. 1115–1145, Nov. 1995. [Online]. Available: https://doi.org/10.1145/227683.227684

[20] J. Soto, "Improved analysis of a max cut algorithm based on spectral partitioning," 2014. [Online]. Available: https://arxiv.org/abs/0910.0504

[21] B. H. Theilman and J. B. Aimone, "Goemans-williamson maxcut approximation algorithm on loihi," in *Proceedings of the 2023 Annual Neuro-Inspired Computational Elements Conference*, 2023, pp. 1–5.

[22] E. Oja, "Principal components, minor components, and linear neural networks," *Neural Networks*, vol. 5, no. 6, pp. 927–935, 1992. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0893608005800899

[23] J. Leskovec and R. Sosič, "Snap: A general-purpose network analysis and graph-mining library," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 8, no. 1, p. 1, 2016.

[24] A. Dubovskiy, T. Criss, A. S. E. Valli, L. Rehm, A. D. Kent, and A. Haas, "One trillion true random bits generated with a field-programmable gate array actuated magnetic tunnel junction," *IEEE Magnetics Letters*, vol. 15, p. 1–4, 2024. [Online]. Available: http://dx.doi.org/10.1109/LMAG.2024.3416091

[25] M. A. Coletti, E. O. Scott, and J. K. Bassett, "Library for evolutionary algorithms in python (leap)," in *Proceedings of the 2020 Genetic and Evolutionary Computation Conference Companion*, 2020, pp. 1571–1579.

[26] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: Nsga-ii," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, 2002.