2019

Design and Investigation of Genetic Algorithmic and Reinforcement Learning Approaches to Wire Crossing Reductions for pNML Devices

Alexander Keith Gunter
University of Mississippi

DESIGN AND INVESTIGATION OF GENETIC ALGORITHMIC AND REINFORCEMENT LEARNING APPROACHES TO WIRE CROSSING REDUCTIONS FOR PNML DEVICES

A Thesis
presented in partial fulfillment of requirements
for the degree of Master of Science in Engineering
Science with Emphasis in Electrical Engineering
The University of Mississippi

by

ALEXANDER GUNTER

December 2018

ABSTRACT

Perpendicular nanomagnet logic (pNML) is an emerging post-CMOS technology which encodes binary data in the polarization of single-domain nanomagnets and performs operations via fringing field interactions. Currently, there is no complete top-down workflow for pNML. Researchers must instead simultaneously handle place-and-route, timing, and logic minimization by hand. These tasks include multiple NP-Hard subproblems, and the lack of automated tools for solving them for pNML precludes the design of large-scale pNML circuits.

In this thesis we investigate potential solutions to the problem of wire crossing reduction in pNML circuits. Although pNML permits 3D architectures, planar designs are still preferred for the ease of fabrication; and reducing out-of-plane nanomagnets reduces risks of fabrication effects. We have found no existing work on this problem in pNML, and existing work for related technologies does not consider variations of the wire crossing problem that are specific to pNML. We present and evaluate two algorithms designed to address this research gap. The first is a genetic algorithm utilizing a multi-chromosome encoding of graph embedding, and the second is a deep reinforcement learning algorithm utilizing a similar encoding and Q-Learning. We also present a naïve NP-time randomized search algorithm for use as a baseline. The presented reinforcement learning algorithm proved unacceptably slow and ineffective, but our genetic algorithm removed significantly more wire crossings than the random search did on the ISCAS ’85 combinational benchmarks.

ACKNOWLEDGEMENTS

I would like to first thank my partner, Wil Adamec, for his constant support and reassurance throughout the past two years, without which I would not have been able to complete this degree. Similarly, I thank my mother, Kim Gunter, my father, Greg Gunter, and my brother, Ian Gunter, for all their advice and encouragement and for giving me an upbringing that enabled me to pursue these studies. I thank my advisor, Dr. Matthew Morrison, for his patience and guidance and for all the opportunities he opened for me over the past four years; and I thank my labmates George Humphrey, Maisha Sadia, and James Haywood for their feedback and friendship. I would also like to thank my committee members, Dr. Yixin Chen and Dr. Richard Gordon, for their time and input on this thesis; and I thank Drs. Dawn Wilkins, Conrad Cunningham, Naeemul Hassan, Sandra Spiroff, and Bing Wei for putting up with all my weird questions. I also thank Dr. Kevin Beach for initially introducing me to QCA and thus giving me the very first nudge towards this project. Finally, I thank the Department of Electrical Engineering, the Department of Computer and Information Science, and the Honors College for funding my studies and providing facilities that have been invaluable to my work.

TABLE OF CONTENTS

LIST OF TABLES

TABLE PAGE

LIST OF FIGURES

FIGURE PAGE

CHAPTER I

INTRODUCTION

Quantum-dot cellular automata (QCA) is a class of emerging post-CMOS technologies which promise ultra-low power consumption, persistent state in the absence of power, and intrinsic pipelining [1-3]. These circuits attain these objectives by encoding binary data in the state of bistable cells which are coupled by fringing electric or magnetic fields [1]. An adiabatic clocking field drives data propagation through the circuit and supplies power gain to prevent signal degradation. One of these technologies, magnetic QCA (MQCA) or nanomagnet logic (NML) implements these cells as nanomagnets [3]. Resultantly, NML is radiation-hard and easy to fabricate compared to most electric QCA implementations.

Perpendicular NML (pNML) is an implementation of NML with irregularly shaped cells which do not require clocking zones [4]. This technology is advantageous because its nanomagnets are planar and polarize perpendicularly, and support ferromagnetic and antiferromagnetic coupling [12]. Each nanomagnet has an artificial nucleation center (ANC) functioning as an input contact. Unlike conventional QCA, pNML clocking fields do not directly require intermediate metastable states due to the ANCs directing data propagation.

Many currently-existing QCA design automation techniques are not applicable due to the irregularly shaped cells and the finer-grained pipelining permitted by pNML. Along with QCA's potential for ultra-low power operation, pNML circuits are inherently radiation hard and may enable 3D architectures that are easier to fabricate and have clocking systems with reduced

clocking tree overhead. Successful design and implementation of pNML circuits would allow for a low-power and reliable solution for high-altitude flight for military, commercial and space applications, nuclear reactors, and deep-well drilling. One NP-Hard optimization problem that remains unaddressed is wire crossing reduction in pNML circuits. This is notable because the problem is almost independent of the design rule set used, but it still has unusual constraints due to the behavior of pNML nanomagnets.

In this thesis, we present our investigation into the potential for genetic and reinforcement learning algorithms to solve this problem. In Chapter II we review the current literature with emphasis on the principles of Quantum-dot Cellular Automata (QCA) and its subcategories of Electric QCA, Magnetic QCA, and perpendicular Nanomagnet Logic (pNML). We also describe the state of electronic design automation for pNML and review topics in genetic algorithms, reinforcement learning, graph embeddings, and the Apache Spark Framework. In Chapter III we derive our approach to the problem including our representation of pNML circuits with graph embeddings. We then describe the software necessary to convert combinational Verilog circuits into pNML circuits. In Chapter IV we present our genetic algorithm by explaining our genome representation of a graph embedding, our algorithm to identify nonplanar edges, how we distribute the algorithm using Spark, and the final results of our experiments. Chapter V does the same with our reinforcement learning algorithm by explaining our representation of the problem as a Markov Decision Process, the topology of our neural network, our training algorithm, and our results for its performance. Chapter VI explains our implementation of a random search algorithm for use as a baseline. We close with Chapter VII where we analyze our results and present future research avenues that could improve upon or supplement our work.

CHAPTER II

BACKGROUND

II.1. Principles of QCA

Quantum-dot Cellular Automata (QCA) is a transistorless processor technology comprised of discrete field-coupled cells [5]. These cells are engineered to have a bistable internal state, usually implemented as an electric or magnetic dipole via ensembles of electric charges or electron spins. These dipoles necessarily generate an external field and respond to changes in that field. Two dipoles can thus interact with each other via their fringing fields. By carefully engineering the cells, we can restrict the number of stable states for the internal dipoles to just two where each state produces a different fringing field. An ambient field can then bias a cell towards one of these states.

QCA circuits are regulated by controlling this ambient field in a manner that lets cells determine the states of their neighbors. We do this with a clocking field that acts to lower the energy barrier between the two states, letting cells switch to the state induced by their neighbors according to a consistent schedule. For implementations with very simple cells, the clocking field serves as a mechanism for controlling the direction data propagates in. Oscillating the clocking field adiabatically minimizes error rates and dynamic energy dissipation. Per-cell power dissipation typically reaches the nano- and picowatt ranges, making QCA an inherently ultra-low power technology [35, 36].

Because electromagnetic fields combine additively, we can implement majority logic using ensembles of cells. Usually these logic gates are 3-input MAJ gates, which output a 1 when at least two of the inputs are 1 as per Table 1. This gate is notable because all three of its inputs are treated symmetrically. Using any one of them as a control signal toggles the gate between a 2-input AND or a 2-input OR gate. QCA implementations also support NOT gates, so they are functionally complete.

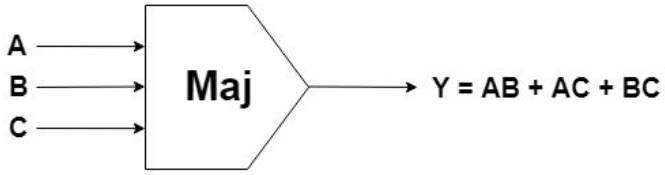


Figure 1. The Majority Gate and Its Boolean Expression

Table 1: Truth Table for a Majority Voter Gate

C	A	B	Y
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

II.2. Electric QCA

Electric QCA (EQCA) is the original formulation of QCA by Lent, Taugaw, and Porod [6]. It uses electric dipoles and electric fields. Generally, its cells are implemented as quantum-dots where electrons may or may not reside. These dots are arranged into a square and are close enough to be tunnel-coupled, so the electrons can tunnel between them. This arrangement is bistable because the Coulomb force between the two electrons drives them to maximize the distance between each other, and this corresponds to the two diagonals of the square. Because of the tunnel-coupling, the cell's state can change in response to the ambient electric field.

These cells are the simple kind that we referred to in Section I.1. They have no internal mechanism for directing data propagation, so the clocking system must control that. Otherwise signals would propagate as a random walk through the circuit and could in fact travel backwards and drastically slow down the circuit's operation [7, 8]. To prevent this, the circuit must be partitioned into clocking zones. These are small contiguous ensembles of cells, usually comprised of only three to five cells, which experience the same clocking signal at the same time. This clocking signal must also be designed such that it forces the cells into a neutral METASTABLE state. Clocking zones are activated in a rolling multi-phase fashion, so a given zone's outputs are always reset while it is reading from its inputs. This prevents signals from propagating backwards and gives EQCA circuits zone-level pipelining. Each zone can only store one signal at a time, but multiple signals can propagate across the circuit's zones at once. This process is illustrated in Figure 3.

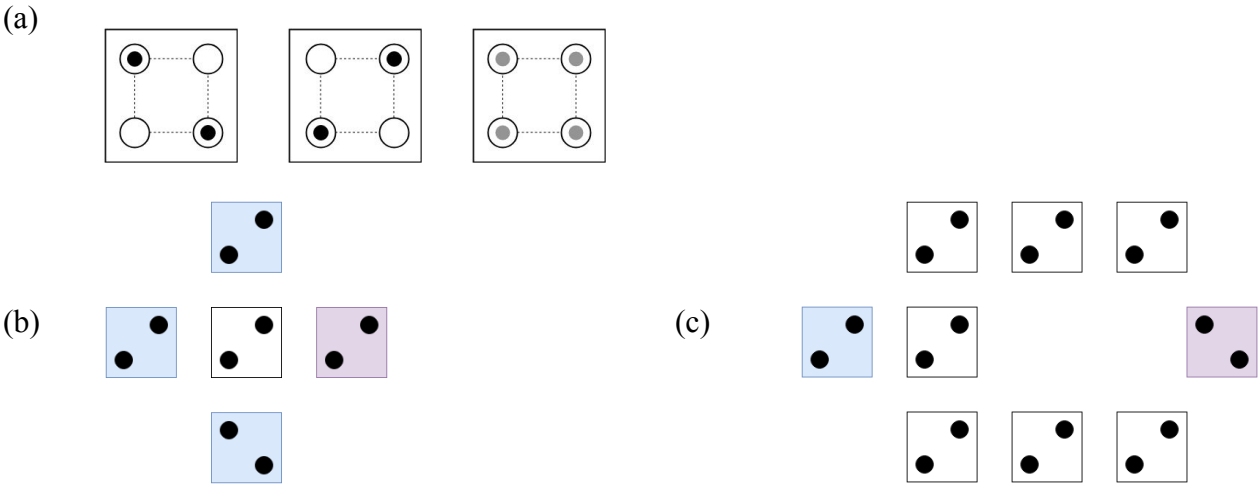


Figure 2. EQCA Cells and Basic Logic Gates. (a) Schematics of EQCA cells. Two are polarized, and the third is in a METASTABLE state. (b) An EQCA majority voter where the inputs are in blue, and the output is in purple. (c) An EQCA inverter.

EQCA is most notable for its small feature sizes and potential switching speeds. This is because currently-proposed designs use cells fabricated from individual molecules or even arrangements of dangling bonds. Atomic Silicon Quantum Dots currently permit the fabrication of cells with dots 2 nm apart with switching frequencies in the GHz range [9]. The theoretical limit for these could feature cells with dots separated by only 2.3 Å and switching frequencies in the THz range [9]. Design rules for EQCA circuits can be relatively simple as well due to the consistency of EQCA cells. The rule set primarily needs to specify the width of and distances between cells. It might also specify the resolution with which cells can be placed, permitting off-center cells. However, these performance-oriented characteristics make EQCA circuits extremely difficult to fabricate. This is both because of the difficulty of placing the cells and the difficulty of creating a clocking system that is both powerful enough to control the circuit but compact enough to fit on the chip.

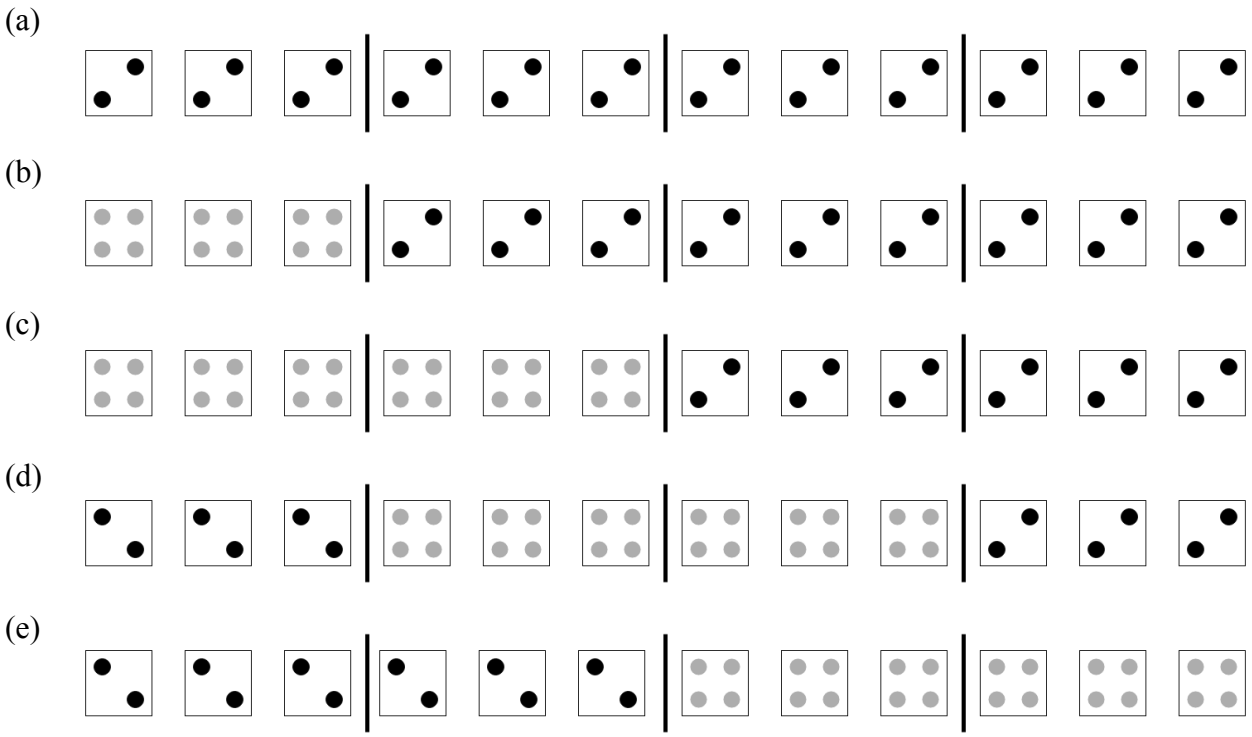


Figure 3. EQCA Clocking. (a) A four-zone wire segment in its initial state with no zones activated. (b) Activate the first zone. (c) Activate the second zone. (d) Activate the third zone and deactivate the first zone. This reevaluates the first zone without reading the second zone, and the wire segment now stores two signals simultaneously. (e) Activate the fourth zone and deactivate the second zone. This reevaluates the second zone by reading the first and without reading the third.

II.3. Magnetic QCA and In-Plane NML

Magnetic QCA, also called nanomagnet logic (NML), is currently separated into two very different implementations. These are called in-plane NML (iNML) and perpendicular NML (pNML), referring to the axis along which the nanomagnets polarize. In both technologies, its cells are implemented as single-domain nanomagnets whose anisotropy drives them to polarize in one of two directions [10]. Nanomagnets can either couple ferromagnetically or antiferromagnetically, so inverters are extremely compact [10, 11]. This in turn means both NML types are more easily modeled in terms of minority logic, and it makes the NAND and NOR gates more compact than AND and OR.

iNML is characterized by the use of rectangular nanomagnets which polarize in the fabrication plane [12]. By convention, nanomagnets are longer along their *y*-axis, so they polarize either UP or DOWN [10]. The coupling between two such nanomagnets depends on their relative position. Using the UP/DOWN convention, vertically aligned nanomagnets will couple ferromagnetically; so a nanomagnet in the UP state will bias its neighbor towards the UP state. Horizontally aligned nanomagnets will couple antiferromagnetically, so a nanomagnet in the UP state will bias its neighbor towards the DOWN state. Inverters are implemented as pairs of horizontally-aligned nanomagnets [12]. iNML minority gates use the same cross-shaped intersection as EQCA majority gates as shown in Figure 4.

Like EQCA cells, iNML nanomagnets do not have an internal mechanism for directing data propagation; so iNML circuits are also partitioned into clocking zones [13]. The main difference here is that the METASTABLE state is a horizontal polarization, so the clocking field must be strong enough to overcome the anisotropy of the nanomagnets. The clocking behavior is otherwise identical to that of EQCA; and its design rules are also very similar to EQCA's. Because the nanomagnets are considerably larger, often in the realm of 50-100 nm long [10, 12], the clocking system is much easier to fabricate than in EQCA. However this size means iNML is much lower performance, offering clocking frequencies in the MHz range [14]. iNML also suffers from the same strong coupling between layout and timing where changing the placement of cells can significantly alter signal timing [10]. While EQCA circuits' reliance on small numbers of charges makes them sensitive to ionizing radiation, iNML's nanomagnets use a large number of electron spins, making them inherently resistant to this.

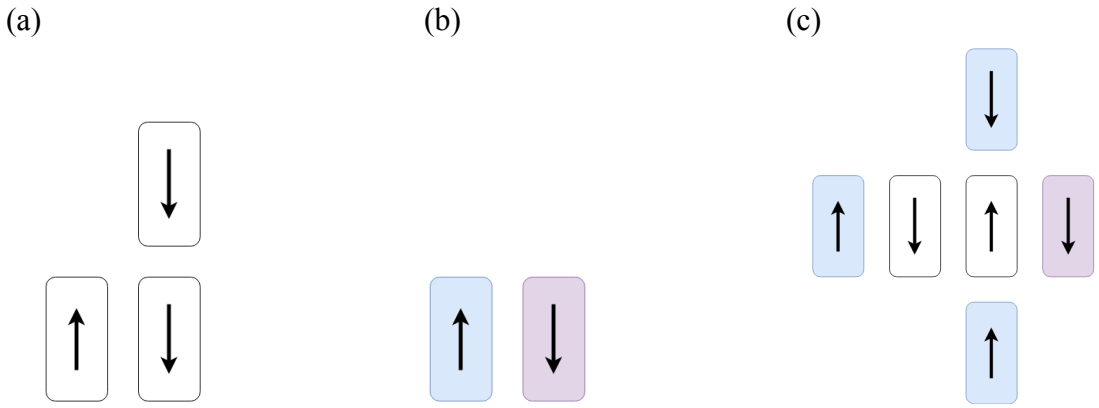


Figure 4. iNML Nanomagnets and Basic Logic Gates. (a) Nanomagnets illustrating iNML coupling. The horizontal pair couples antiferromagnetically, and the vertical pair couples ferromagnetically. (b) A pair of nanomagnets implementing an iNML inverter. (c) An iNML minority gate.

II.4. Perpendicular NML

pNML circuits are notably different from both EQCA and iNML circuits. Their nanomagnets are fabricated from layered nanofilms which produce structures that polarize perpendicularly to the fabrication plane [15]. This perpendicular anisotropy is largely independent of the magnetic domain's shape, so pNML nanomagnets are not restricted to a single shape such as a square or rectangle. While iNML nanomagnets change states very smoothly and without splitting into multiple domains, pNML nanomagnets nucleate [16]. When the ambient field is strong enough and polarized in the opposite direction, regions of the nanomagnet swap polarization, and the inversion propagates throughout the nanomagnet as a domain wall. To control this nucleation, pNML nanomagnets are fabricated with an Artificial Nucleation Center (ANC) [17]. This is created by irradiating a miniscule region of the nanomagnet to slightly intermix the layers of its substrate. This reduces the anisotropy of that

region, making it more sensitive to ambient magnetic fields than the rest of the nanomagnet. The ANC thus becomes the seed point for nucleation.

The ANC makes pNML's behavior unique from other QCA implementations because they make every nanomagnet behave as a diode [18]. By calibrating the clocking field and the anisotropy of the ANCs, we can make pNML circuits where the ambient magnetic field is never strong enough to nucleate nanomagnet bodies. When correctly calibrated, ANCs only nucleate when both the nearby nanomagnets and the clocking field induce the opposite polarization. That is, pNML nanomagnets only change polarizations when that is the logically correct state, and the clocking field is polarized in that direction. This means that the clocking system is not responsible for controlling the direction of data propagation; that is handled internally by the cells themselves. Then the circuit does not need to be partitioned into clocking zones, and the clocking signal does not need to push the nanomagnets into a METASTABLE state.

pNML circuits thus have six notable advantages over iNML circuits. (1) They have cell-level pipelining because each nanomagnet behaves like a diode, so their states are all evaluated independently. (2) Each nanomagnet behaves like a register because their states are never erased by the clocking signal. This produces advantage (3), which is that pNML minority gates with an even number of inputs have well defined behavior in the event of a tie. The tie means that the ambient field produced by the inputs is near-zero, and the nanomagnet does not change its state because the clocking field alone is not strong enough to nucleate its ANC. (4) Because the nanomagnets can have nearly any shape, we can route signals without altering the circuit's timing so long as the clocking field's frequency gives every nanomagnet enough time to invert. (5) Because the clocking field is a global field that just oscillates between UP and

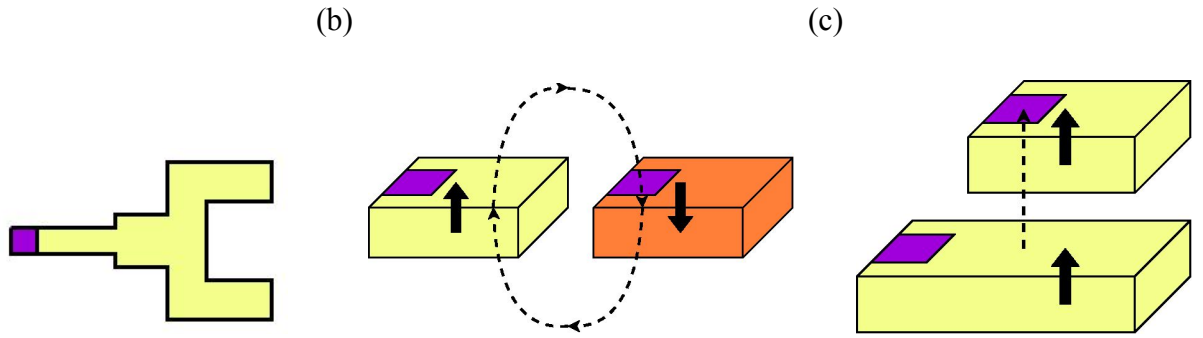


Figure 5. pNML Nanomagnets and Coupling. (a) A schematic of a simple nanomagnet as presented in [18] with the ANC colored in purple. (b) Coplanar pNML nanomagnets always couple antiferromagnetically (c) pNML nanomagnets couple ferromagnetically when the input's body is directly below the output's ANC.

DOWN, the clocking circuitry is considerably simpler. Even if a circuit chooses to have clocking zones, these zones can be the size of an entire component; so the clocking system does not have to be nearly as densely packed as the QCA cells. (6) Because the nanomagnets are made from nanofilms, pNML permits multiple fabrication planes and create 3D architectures. Nanomagnets which reside in the same plane or are not vertically aligned will couple antiferromagnetically, while vertically aligned nanomagnets in separate planes will couple ferromagnetically.

The main disadvantage of pNML compared to iNML is that EDA tools have to synthesize the geometry of each nanomagnet because pNML does not specify any particular nanomagnet shapes. This makes design rules for pNML inherently more sophisticated than those for EQCA and iNML as they have to be capable of generating compact geometry and must support pathing but without compromising circuit reliability. This also introduces a tradeoff between clocking frequency and circuit latency as permitting longer paths means domain walls require more time to reach their outputs.

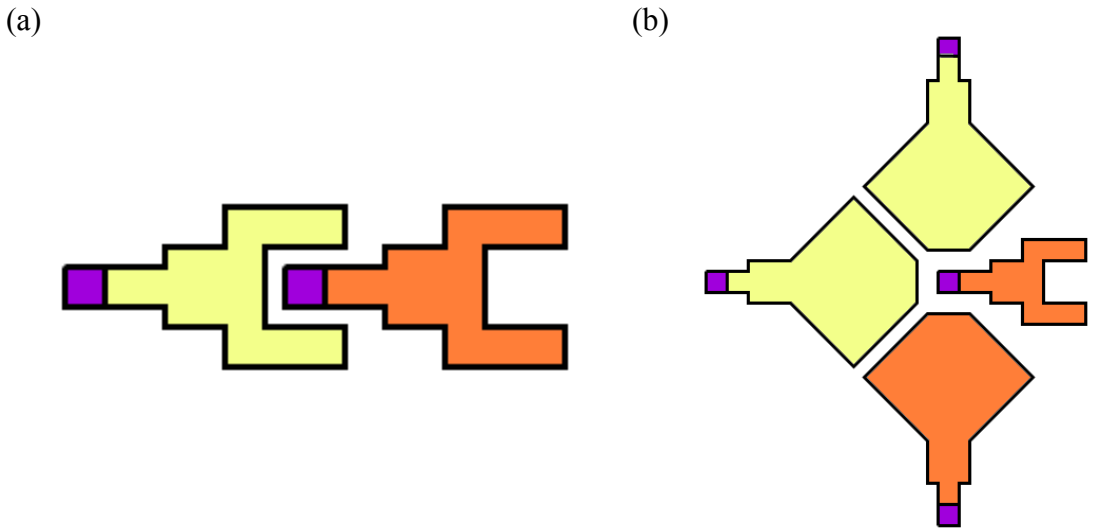


Figure 6. pNML Basic Logic Gates. (a) A pNML inverter. (b) A pNML minority gate.

II.5. pNML EDA

Compared to other QCA implementations, work on design automation of pNML circuits is sparse. Prior work in general QCA that pNML inherits primarily consists of majority logic synthesis. One of the first scalable automated approaches presented in [19] begins by decomposing the logic network into gates of at-most three inputs and uses Karnaugh maps to either identify majority voters or decompose gates into multilayer majority voter networks. This approach has been augmented several times to further reduce the number of majority gates [20] and utilize majority gates with more than three inputs [21]. Alongside this approach have been genetic algorithms as in [22] and manipulation of tree-like data structures as in [23] and [24]. The Majority-Inverter Graph presented in [24] is especially notable for its use of a well-defined algebra which has yielded significant reductions in delay and area over other tools.

The most fundamental tool for pNML is the Object Oriented MicroMagnetic Framework (OOMMF). The framework performs high-fidelity simulation of magnetic materials [25]. This

enables the creation and exploration of novel magnetic technologies at small scales, provided the engineer knows the material's magnetic properties and geometry. Ju et al. used this tool in [16] to demonstrate the switching and coupling behavior of Co/Pt nanomagnets for use as pNML cells.

Despite its accuracy and versatility, OOMMF only immediately helps characterize a given geometry and does not include tools to search for geometries which correctly perform a given function. To this end, Notre Dame has presented a machine-learning approach based on least squares regression [11]. In this approach, the algorithm is given an initial circuit design which nearly implements a known function. The algorithm then uses OOMMF to simulate the circuit and compares the simulated outputs to the expected ones. The algorithm then uses linear regression to correlate geometric parameters to the correctness of the circuit. This correlation becomes a heuristic for searching the physical design space.

The closest existing thing to a traditional top-down workflow for pNML is the MagCAD program [26], which supports the manual placement of pNML nanomagnets. The tool can export these designs to its sister-tool ToPoliNano which can simulate the circuit using VHDL. Although ToPoliNano can synthesize VHDL descriptions of iNML circuits, it currently cannot do this for pNML. It is also unclear whether existing scalable simulation techniques enable analysis of timing concerns specific to pNML.

II.6. Previous Work in Crossing Reduction in QCA

When designing circuits, we usually want to work in a top-down fashion. We want to specify high-level behavior and have design automation software convert that specification to an

optimized circuit design ready for fabrication. We should be able to verify that the design behaves as expected, verify that the expected behavior actually completes the desired task, and verify that the design satisfies any performance requirements we may have.

This process begins with the circut's behavior and high-level architecture specified with a hardware description language (HDL) such as Verilog. This is what the design engineer primarily creates and maintains. This HDL must then be synthesized into a netlist of wires and logic gates. Then the EDA software should simplify this netlist to minimize the number of components required, removing redundant components or improving suboptimal structures. Once the logic has been minimized, it must be synthesized into physical components which occupy space. These components must be placed on the substrate, and the wires connecting them have to be routed such that data signals do not interfere with each other. These components must also be arranged in a manner that reduces the area and difficulty of fabricating the circuit. Because physical signals take time to propagate, all components must also be arranged such that all signals are correctly synchronized and propagate quickly enough to meet performance requirements.

This entire process entails solving multiple NP-Hard problems, such as logic minimization and place-and-route with constraints. The process also requires knowledge about the specific technology being used as this directly impacts the nature of all geometric and timing constraints. Thus, although many such workflows exist for a multitude of different transistor-based technologies, these workflows cannot be directly ported to pNML. One of the subproblems of the place-and-route stage is the minimization of wire crossings in the circuit. These structures require either 3D structures that are hard to fabricate or complex arrangements

of logic gates that reduce performance. This subproblem is itself an NP-Complete problem, even when taken in isolation from the other NP-Hard tasks in the top-down workflow.

There has been work on crossing reduction in EQCA circuits, and iNML circuits by extension; and some of this is applicable to pNML. Node duplication techniques as in [37] duplicate substructures in the circuit to bypass obstructions. Purpose-built gates like the UQCALG presented in [38] offer the flexibility to construct combinational structures that can be fabricated with very few crossings. Such gates are used by algorithms like the one presented in [39]. There are also heuristic methods based on channel routing [40]. The problem with the first three is that they only synthesize logic in a manner that reduces the minimum-possible number of crossings. Another algorithm still has to perform the placement. The channel routing methods take care of this, but they do not currently account for the unusual geometric constraints that come with pNML.

II.7. Genetic Algorithms

Genetic algorithms are a class of algorithms inspired by the mechanics of biological evolution [27]. They are fundamentally characterized by the representation of candidate solutions in a way that permits the recombination of candidates into new candidates. They also presume that, on average, the random recombination of good candidates yields more good candidates. These candidates are encoded as genomes which are comprised of one or more chromosomes. Each chromosome is a sequence of tokens or values which represent specific characteristics of the candidates solution. At each iteration, the algorithm takes a population of genomes and chooses pairs of them using a selection algorithm and according to a fitness function. These pairs

are recombined to create a new population, and those new genomes are then tweaked with a mutation algorithm.

Decoding a genome into a form usable outside the genetic algorithm should be more or less efficient, certainly polynomial-time; but priority is given to optimizing the functions for fitness evaluation, selection, recombination, and mutation. The fitness evaluation function is responsible for scoring the quality of a genome and is used by the selection algorithm. Usually the algorithm tries to maximize this akin to "survival of the fittest," but it's possible to minimize it instead. Given a population of genomes, the selection algorithm uses the fitness function to create a distribution of genome-pairs from the population. It then selects pairs of genomes according to this distribution for recombination.

Recombination, also called crossover, is the process of intermixing two genomes to create a new one. A multitude of algorithms for this exist, each being applicable to different kinds of problems. For example, problems in which each chromosome element is an independent parameter might randomly choose which parameters to inherit from which parent. Combinatorial, order-sensitive chromosomes might instead be recombined using an order-preserving algorithm. The end result is a new genome which inherits characteristics of both parent genomes, much like in biology.

Genetic algorithms also pull the concept of mutation from biology. These algorithms are applied to genomes after they've been created from recombination. These are designed to slightly perturb the population, making the process less prone to converging to local optima. Examples of mutation algorithms include adding small random deviations to chromosome

elements or swapping the order of two elements. Usually the mutation algorithm is randomly selected from a set of such algorithms.

There are a variety of ways to begin and end the evolutionary process. Usually the initial population is generated according to a distribution suitable to the problem. The termination condition can be based on any metric for the procedure's performance. This can be as simple as the population satisfying some fitness threshold, or it can be a limit on the number of iterations. It's also possible to track the quality of the population and stop when the rate of improvement falls below a given threshold. The final output can be decided in a number of ways too, such as the best solution in the last generation or the best across all generations. It could also be selected from the last generation according to a distribution based on the fitness values. The output could even be the last generation as a whole, and a problem-specific process can reduce that to a final solution.

II.8. Reinforcement Learning Algorithms

Reinforcement learning algorithms begin with an Environment/Actor model of the problem [28]. The Environment has an internal state, and the Actor can choose an action to apply to the state. Given the current state and chosen action, the Environment uses a Markov Decision Process (MDP) to determine the next state, as well as a reward. The Actor takes this new state and reward and chooses a new action, and the cycle continues until the system reaches a terminal state. This process produces a sequence of state-action-reward triples according to a distribution. The goal of reinforcement learning is to train the Actor to choose actions which maximize the total reward received.

The environment is usually derived from the problem, often by "gamifying" it. It requires a state representation that can be efficiently modified based on chosen actions. This might be a game board with movable pieces placed on a grid, or it could be a distribution that never changes. It also must dispense rewards based on both the current state and chosen action, so the Actor has goals to achieve and can identify them. For example, moving a board piece left might capture a resource and yield a high reward; or it might push the piece out of bounds and yield a negative reward. Because actions are applied over time, the system can also have a concept of a future reward; and this introduces a trade-off between immediate and future rewards. Typically the Environment encodes this as a hyperparameter called the reward discount, which is a scaling factor between 0 and 1 applied to a reward based on how many steps away it is.

Actors in reinforcement learning algorithms usually have two critical components: a policy $\pi(A \mid S)$ and a value function. The policy is responsible for choosing an action based on the total reward the Actor can expect to receive afterwards, and it is in fact a distribution of actions.

$\pi(A \mid S)$ is thus the probability of choosing action A given state S. The value function can either be a state-value function $V_\pi(S)$ or an action-value function $Q_\pi(S, A)$. The state-value function predicts the expected total reward that the Actor will receive if it begins in a particular state S and follows policy π afterwards. The action-value function predicts the expected total reward if the Actor begins in state S, takes action A, and then follows policy π from then on. Each of these options has advantages worth considering. Specifically, the state-value function typically requires less memory to store because it has a smaller domain; but the action-value function is easier to modify because doing so does not require checking all actions applicable to

S. Reinforcement learning is about using the interdependence of the policy and value function to iteratively train a skilled Actor.

Initially the value function will be highly inaccurate and produce bad predictions for expected returns. This will drive the policy to choose bad actions very frequently. However, in doing so, the Actor will sample rewards from the Environment; and it can use those samples to update its value function's predictions. This in turn can alter the policy, so it chooses better actions. If the value function's predictions for each state or each state-action pair are calculated independently, such as by being stored in a lookup table, then this process will eventually converge to optimal behavior by the Policy Improvement Theorem [28].

One of the most commonly used training algorithms is Q-Learning. This approach uses an action-value function and takes the learning rate α as a hyperparameter, alongside the reward discount γ from the Environment. At each step, the Actor begins with the current state S and the current value function Q. It derives a policy π from Q and chooses an action A according to $\pi(S)$. It passes A along to the Environment and receives the reward R and the new state S'. To update Q, it calculates and scales the error in its prediction and then adds that error to its current estimate. This update function is

$$Q(S,\ A) \leftarrow Q(S,\ A)\ +\ \alpha[R + \gamma\ max_a Q(S',\ a)\ -\ Q(S,\ A)] \tag{1}$$

where γ is the reward discount. Thus Q is pushed towards the actually received. In the limit where every state-action pair is tested infinitely many times, Q will exactly predict the expected total reward for every state-action pair.

II.9. Graph Embeddings

A fundamental component of this work is the graph embedding, which can be thought of as the drawing of a finite graph on some surface. Ordinarily we encounter these any time we see a graph drawn on paper, but the idea can be generalized using topology and combinatorics. Depending on the properties of interest, different representations may be more useful.

We can begin deriving these representations by identifying unhelpful information stored in the naive representation of an embedding - a drawing on the coordinate plane. This representation, which we call a spatial embedding, is highly detailed with each vertex being assigned to a unique point. Each edge is assigned to a simple curve whose endpoints are the points representing the edge's vertices, and the curve's interior points never represent vertices. For this section, we also assume curves that intersect only intersect at most at one point. This simplifies our analysis without loss of generality.

We can see that this representation stores more information than necessary for identifying intersecting edges because we can move vertices slightly without changing the crossings in the graph. All that changes is the position of the vertex and some distances between edges. Thus we can infer that some equivalence class of embeddings exists such that all of its members have the same crossings as the embedding we have on-hand. A representation of that equivalence class should be simpler and more well-suited for identifying edge crossings.

Topology provides the tools for identifying this equivalence class. A plane is a surface of genus 0, as is the surface of a sphere [29]; and we can map a circular region of a plane onto the surface of a sphere with a continuous function. Let an embedding in a coordinate plane be bounded by a circle of radius ε centered at the origin such that no vertex or edge touches the

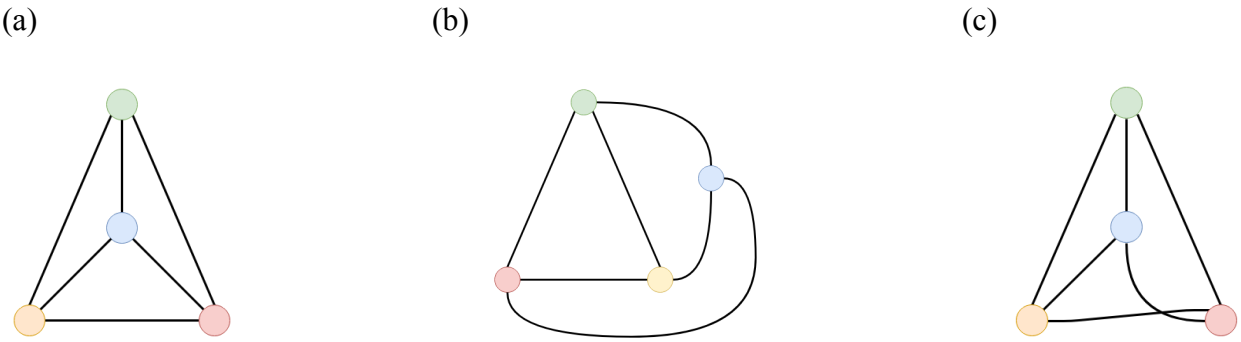


Figure 7. Embeddings of K_4. (a) The straight-line planar embedding of K_4. (b) An alternative embedding that is equivalent to the first one. This embedding is made by transforming the first embedding with f, rotating the resulting sphere embedding by approximately 180°, and applying f^{-1}. (c) An embedding with an unnecessary crossing. The crossing is unnecessary because the yellow-red edge could be routed through the outer face.

circle. Then all points in the circle have polar coordinates of the form (r, θ) for $r \in [0, \varepsilon)$. Then we can map each point (r, θ) to the surface of a sphere of radius ε using the function

$$f(r, \theta) = (2\sqrt{r(\varepsilon - r)}, \theta, 2r) \quad (2)$$

where the right-hand side is in cylindrical coordinates. For all cylindrical coordinates (r', θ, z) where $z \in [0, 2\varepsilon)$, we can invert this with Equation 3.

$$f^{-1}(r', \theta, z) = (0.5z, \theta) \quad (3)$$

This spherical representation makes many transformations more clearly irrelevant to edge crossings. For example, rotating the spherical embedding has no effect on the relative position of any vertices or edges and thus cannot affect crossings. Continuously expanding or contracting a region on the sphere similarly has no effect on crossings. As long as these transformations don't move a vertex or edge to the coordinate $(0, 0, 2\varepsilon)$, we can map the spherical embeddings to planar embeddings and obtain planar drawings that are drastically different but have the same crossings.

We observe that, sufficiently close to a vertex, the cyclic order of its incident edges is invariant under these continuous transformations. Thus we can instead use a combinatorial representation of an embedding rather than a spatial one. One such representation is the rotation system [30]. To construct a rotation system, we first split the graph's edges into darts, or half-edges. In the case of undirected graphs, these are simply the two orientations an edge can have, (a, b) or (b, a) for vertices a and b. Thus we map each edge e_i to a pair of darts $(d_{i,1}, d_{i,2})$. Then we construct the permutation θ such that such that $\theta(d_{i,1}) = d_{i,2}$ and $\theta(d_{i,2}) = d_{i,1}$ for each edge e_i. We then construct the permutation σ according to the clockwise ordering of darts about their shared origins. That is, for dart $d = (a, b)$, $\sigma(d)$ returns the dart clockwise to d about its origin vertex a. The rotation system is defined as the pair (θ, σ) and completely characterizes an equivalence class of spatial embeddings.

Given a rotation system, we can formally define orbits in an embedding. We have already encountered vertex orbits. They are the cycles in σ and, as stated before, are the cyclic orderings of darts about shared origins. Face orbits are similarly defined as the cycles in the composite permutation $\sigma\theta$. In the case of straight-line planar embeddings with no crossings, face orbits correspond to the counter-clockwise cyclic orderings of darts about the polygons on their left sides. Each rotation system also has an inverse system which reverses these orderings. This can be thought of as either reflecting a spatial embedding or swapping the convention for vertex orbits to be counter-clockwise orderings. This inverse system features the same crossings.

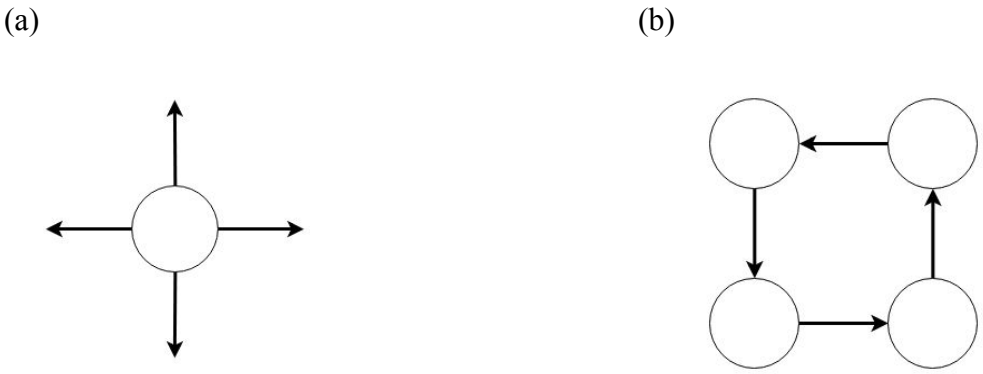


Figure 8. Orbits in Embeddings. (a) A vertex orbit. (b) A face orbit.

II.10. The Apache Spark Framework

The Apache Spark Framework is a framework for distributed computing on the Java Virtual Machine and is based on Hadoop's MapReduce model [31]. The framework's fundamental data structure is the Resilient Distributed Dataset (RDD). This is an ordered list of elements that is split into partitions. These partitions are distributed to the available Spark compute nodes. Each partition is itself ordered, as is the list of partitions. Spark operates on RDDs using transformations, filters, and aggregators. When these operations are commutative and associative, compute nodes can operate independently and asynchronously. Such operations can thus scale extremely well with minimal networking overhead. Spark also supports distributed versions of more complex algorithms such as sorting, grouping, products, and filtering unique elements. These operations require considerably more overhead because they trigger shuffle operations, requiring all nodes to share and trade partitions with each other; but this greatly expands the scope of algorithms supported by Spark.

Spark also has another data structure built on top of the RDD, called the Dataset. This represents an unordered collection of elements, although it technically has an indeterminate order

due to its basis on the RDD. The main advantage of this data structure is its SQL-like API that lets programs process data in the same manner as a relational database. Thus its elements are organized into rows and labelled columns. The Dataset hides most of the implementation details of the RDD but automatically optimizes the processing pipeline and simplifies the API.

To achieve the best possible optimization, Spark uses lazy evaluation and task pipelines to build RDDs. Thus, while the programmer treats an RDD or Dataset as a collection of elements, the instance itself will actually be encoded as the series of operations which compute it. These operations, or tasks, are arranged into a directed acyclic graph (DAG) according to their dependencies and divided into fully parallelizable execution stages. This makes Datasets consumable structures that are computed from scratch whenever needed, offering performance benefits in cases where the Dataset is needed only once. Spark also provides a caching feature in case the Dataset is needed multiple times. When computing the contents of a Dataset using a DAG, Spark will substitute and rearrange tasks to minimize the networking overhead and maximize parallelization. This is a feature currently unique to Spark and makes the MapReduce model far more accessible, applicable, and effective. Programmers simply have to design their algorithms around a standard relational database structure, and they mainly need to be mindful of which operations consume input Datasets and trigger shuffle operations.

CHAPTER III

PROBLEM ANALYSIS AND REPRESENTATION

III.1 Representation of pNML Wire Crossing Reduction

Given the variability of pNML design rules, we must inspect existing research for common characteristics and restrictions on pNML geometry. Our first observation is that all designs published to date have exactly one ANC per nanomagnet. This makes sense because conflicting signals, combined with process variations, would likely render the magnet's behavior indeterminate. Such designs would require a guarantee that multi-ANC nanomagnets would never receive conflicting signals. Verifying this guarantee would in turn require an NP-Hard analysis of the relevant circuit components. Thus we can assume that future pNML design rules will restrict nanomagnets to one ANC.

Because of this restriction, we can relate pNML circuits to transistor-based digital circuits by treating ANCs as register-like components connected by wires. Specifically, we model each ANC as a threshold gate with an output register whose state can persist between clocking cycles. This gate can have as many input signals as the design rules permit, and those signals may or may not be inverted based on the magnetic coupling between an input magnet and the ANC. In this model, the wires are the bodies of the nanomagnets, and they only transmit data in one direction as per pNML's behavior. This means we can model pNML connectivity as a directed graph where vertices represent ANCs, and a vertex's outbound edges represent the corresponding nanomagnet body. Then a planar embedding of this graph represents the physical

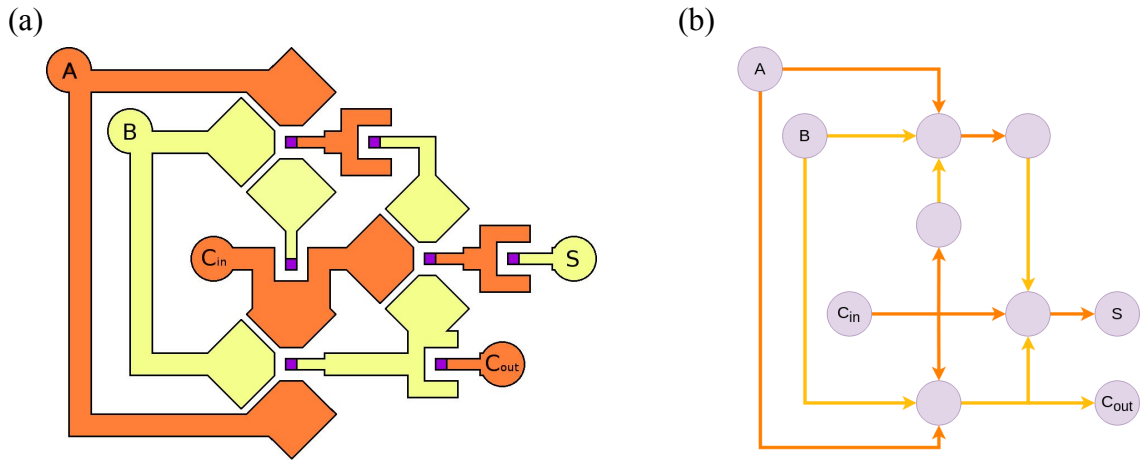


Figure 9. Digraph Representation of a Planar pNML Full-Adder. (a) The physical circuit design presented in [18] with ANCs in purple and nanomagnet bodies in brown and yellow. (b) A digraph representation of the same circuit.

paths along which the nanomagnets transmit data, and a design rule set can synthesize geometry along those paths. Then we can reduce pNML wire crossings by reducing edge crossings in a planar graph embedding.

Because we have related the circuit's geometry to the graph embedding's edges, we incur two variations on the standard crossing reduction problem. The first is a relaxation on the problem: edges with a common origin can cross or even overlap. This is because they correspond to the same nanomagnet body, and a nanomagnet cannot intersect with itself. Then our algorithm must either avoid embeddings which have such crossings, or it must not count them towards the total number of crossings.

The other variation is an unusual restriction that is our motivation for exploring non-standard approaches to wire crossing reduction. Inspecting existing pNML designs reveals that ANCs are almost completely surrounded by their inputs, regardless of the number of inputs. There is only one direction the ANC's body can extend in before branching out to connect to its outputs. This is because the input magnets must occupy a large area near the ANC to produce a

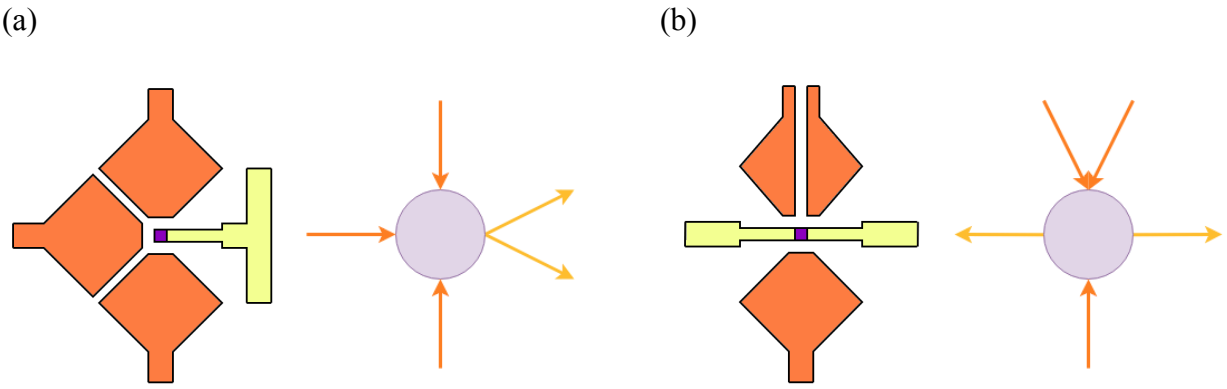


Figure 10. Allowed and Disallowed Nanomagnet Geometries. (a) A permitted structure and its associated embedding. (b) A structure that is disallowed because the outbound edges are not adjacent to one another in the vertex orbit.

strong enough signal to influence it. Altering this restriction to permit bidirectional domain wall propagation is conceivable, but this would require making the ANC more sensitive and thus less robust. It would also be difficult to reliably make all inputs have the same signal strength as inputs might be partitioned into clusters of different sizes. The result is that edges with a common origin in a permitted graph embedding must all occupy a contiguous subsequence of their origin's orbit, as illustrated in Figure 10.

III.2 Verilog Parser and Netlist

The lowest-level circuit representation for the presented program is the Verilog file. This encodes a description of the logical behavior of the circuit. For now we limit these files to very basic syntax that includes wires, buffers, inverters, and n-input AND, NAND, OR, NOR, XOR, and XNOR gates. This is enough to process the combinational ISCAS Verilog benchmarks [32].

To load Verilog files, we first parse them with an external C++ program. This program uses Ben Marshall's Verilog parser [33] to construct the netlist specified by the Verilog file. The parsing program works by identifying module Wires and then Gates. Wires can either be input

(a)

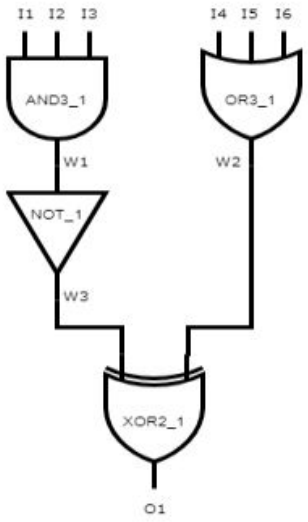
```
module SimpleCircuit (
I1, I2, I3,
I4, I5, I6,
O1
);

input I1, I2, I3;
input I4, I5, I6;
output O1;
wire W1, W2, W3;

and AND3_1 (I1, I2, I3, W1);
or OR3_1 (I4, I5, I6, W2);
not NOT_1 (W1, W3);
xor XOR2_1 (W3, W2, O1)

endmodule
```

(b)



(c)

```
NUM_WIRES
10
WIRE_LIST
0 I1 1
1 I2 1
2 I3 1
3 I4 1
4 I5 1
5 I6 1
6 O1 3
7 W1 2
8 W2 2
9 W3 2
NUM_GATES
4
GATE_LIST
0 AND3_1 1,2,7 0 2
1 OR3_1 4,5,8 3 5
2 NOT_1 9 7 1
3 XOR2_1 8,6 9 6
```

Figure 11. Verilog Sample Files. (a) A sample Verilog file specifying a series of non-trivial gates. (b) The Verilog Netlist derived from the sample file. Integer IDs for wires and gates are omitted for legibility. (c) The .vgraph file for ISCAS benchmark c17.

ports, output ports, or internal wires; and the parser identifies Wire types. Gates can be any of the eight types listed above and can have multiple input Wires and one output Wire. The parser also retrieves the names for all Wires and Gates and records which Wires are connected to a given Gate. After retrieving this labelling and connectivity information, it assigns a unique integer ID to each Wire and to each Gate. It then writes the netlist to a file using our .vgraph format illustrated in Figure 11. It consists of two lists, one specifying the set of Wires and the other specifying the set of Gates. Each wire is encoded as a space-separated triple consisting of its ID, name, and an integer denoting its type. A gate is similarly encoded by its ID, name, its list of input wires as a comma-separated list of IDs, its output wire ID, and an integer denoting its type. This format is designed to be trivial to parse in any language using elementary file iteration and string operations.

These .vgraph files encode Verilog Netlists and are the proper entry point for our algorithms. To load one of these, we iterate over its lines, split each into its component tokens, construct a Wire or Gate from these tokens, and add each constructed instance to either the list of Wires or the list of Gates. These two lists comprise a Verilog Netlist and contain all the information necessary to analyze and characterize the circuit.

III.3 Magnet Layout

The Magnet Layout class sits on top of the Verilog Netlist. It is specifically designed to represent the connectivity information of a pNML circuit. This includes the list of magnets, the magnets each outputs to, the states each magnet can polarize in, and whether two coupled magnets align ferromagnetically or antiferromagnetically. This information is independent of the design rule set and is sufficient to characterize the logic and timing of the circuit.

This data structure is organized similarly to the Verilog Netlist. Constructing it requires a list of Magnets and an adjacency list specifying their connectivity. Each magnet has a unique integer ID, a state of 1 or -1, to represent UP or DOWN polarization, and a set of permitted states {-1}, {1}, or {-1, 1} to denote whether its state can change or not. Magnets with only one possible state are constant-state ancillary inputs to the circuit. The adjacency list for the magnets' connectivity consists of magnet-ID pairs assigned to integer values. Positive values denote ferromagnetic coupling, and negative values denote antiferromagnetic coupling. The magnitude of the value corresponds to the weight of the input magnet's signal in the threshold gate function.

Using this information, we can identify circuit features. Circuit inputs are magnets which have no incoming edges in the adjacency list. Variable circuit inputs are circuit inputs with

permitted states {-1, 1}, and ancillary circuit inputs have the permitted state {-1} or {1}. Circuit outputs are the inverse and are magnets which have no outgoing edges in the magnet layout. Wires are paths in the graph specified by the adjacency list, and inverteres are edges with negative weights. Threshold gates are magnets which have multiple inputs, and a threshold gate might also be a circuit output.

To create a Magnet Layout from a Verilog Netlist, we have to use a synthesis algorithm. This is the point where a pNML EDA workflow could apply logic minimization. We use a very simple one that naively maps Verilog logic gates to majority voter logic and removes duplicate signals. It uses uses a standard hierarchical approach where high-level structures described by the Verilog Netlist are reduced to low-level pNML ensembles and connected to each other. At the lowest level is the Magnet. We define one Magnet for each Wire defined by the Verilog Netlist, and a Wire and its corresponding Magnet share the same ID. This lets us easily compare a Verilog Netlist and its corresponding Magnet Layout. Alongside this is the *buildSimpleWire()* function, which connects two given Magnets with a weight of either 1 or -1. All logical operations can be decomposed into a series of Magnet instantiations and connections.

To create a pNML minority gate, we use the *buildMinorityGate()* function. This accepts three input Magnets and one output magnet. All three inputs are connected to the output with edge weight -1 to denote inverting connections. We also define the *buildHalfAdder()* function which creates a half-adder based on the design presented in [18]. This function accepts three input Magnets and an output Magnet as well, but it has to instantiate five additional Magnets for internal connections. These are all given unique IDs, and the nine magnets are connected using minority gates and simple wires.

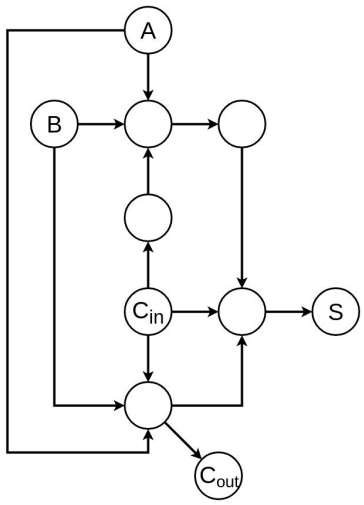


Figure 12. Magnet Layout for a Full-Adder. The full-adder presented in [18] and shown in Figure 9, represented as a Magnet Layout. Edge weights are not shown, but all of them are -1. The circuit inputs are assumed to have ANCs.

The *buildSimpleWire()*, *buildMinorityGate()*, and *buildHalfAdder()* functions for the basis for synthesizing the Verilog Netlist's logic. Buffers and inverters are translated into calls to *buildSimpleWire()*. 2-input NAND and NOR gates are minority gates with ancillary inputs, and 2-input XOR and XNOR gates are half-adders with ancillary inputs. These ancillary inputs do not correspond to wires explicitly defined by the Verilog Netlist, so we instantiate them as needed with unique IDs. 2-input AND and OR gates can be created by adding a new magnet after a NAND or NOR gate, connected with weight -1.

To construct n-input gates, we use the standard pairing of 2-input gates to create a cascade of $O(\log n)$ layers. This is also where we apply some basic logic reduction by inspecting the gate's inputs for duplicate wire IDs. Removal of these duplicates is necessary to ensure that we do not assign multiple conflicting weights to the same edge in the Magnet Layout, and it has the added benefit of reducing the size of the gate. We define specific functions for n-input AND, NAND, OR, NOR, XOR, and XNOR operations.

To convert a Verilog Netlist to a Magnet Layout, we thus begin by instantiating a Magnet for each Wire. We then iterate over the Gates in the netlist. Each Gate stores its type, a list of input Wires, and an output Wire. Buffers and inverters have one input and are mapped to the *buildIDENT()* and *buildNOT()* functions respectively. These are simply wrappers around the *buildSimpleWire()* function. The remaining gate types are n-input gates where *n* is the length of the list of unique inputs for the gate. Each operation is mapped to the associated n-input builder function where duplicate signals are removed, and additional Magnets are created as needed. The resulting list of Magnets and adjacency list is then used to instantiate a Magnet Layout.

III.4 Graph Encoded Map

Rotation systems are compact, but they are somewhat removed from graph embeddings and make it difficult to structurally modify the embedding. They also obfuscate "unnecessary" crossings like the one in Figure 7c. The Graph Encoded Map (GEM) is one solution to this. The GEM contains the same information as a rotation system, but it explicitly stores structural features by simultaneously representing the rotation system's permutations θ, σ, and $\sigma\theta$.

We begin constructing a GEM from a spatial embedding by splitting each edge into two darts. We then equip each dart with two nodes drawn near the origin, one on the left and the other on the right. Thus each edge is equipped with four nodes where two are positioned at each vertex and two on either side of the edge. Each of these nodes is equipped with three colored edges. The first edge, colored red in Figure 13, crosses the parent edge to connect to the other node associated with the same parent dart. The second, colored blue, connects to the node that is in the same face as the parent edge but is associated with the other dart. The third edge, green,

connects to the node associated with the same vertex and the same face. If that vertex only has the given edge in its orbit, then the green and red edges will connect to the same node. Otherwise the green edge connects to a node associated with an edge adjacent in the vertex orbit. To traverse a vertex orbit, we begin at a node in the orbit and traverse the cycle that alternates between green and red. Traversing the cycle that alternates between green and blue will instead traverse a face orbit. Thus we can reconstruct a rotation system from a GEM by traversing the vertex orbits and extracting the sequences of darts encountered.

To implement a GEM with object-oriented programming, we define a high-level object called an Encoding Edge. This encodes a directed edge in a digraph and stores references to its origin and destination vertices. It also stores pointers to its two constituent Darts, as well as the four nodes that represent it in the GEM. We call these Encoding Nodes. These nodes are referred to as the left-origin node, right-origin node, left-destination node, and right-destination node according to their positions in the GEM relative to the Encoding Edge's direction. A Dart also stores references to its origin and destination, except one Dart in an Encoding Edge has these reversed due to its pointing in the opposite direction. The Dart pointing in the same direction as the Encoding Edge is called the origin-dart, and the other is called the destination-dart. The Dart also has a reference back to its parent Encoding Edge, so we can retrieve its sibling Dart or the Dart pointing in the same direction as its parent Encoding Edge. Darts also store references to the Encoding Nodes in the same manner as an Encoding Edge, except that the nodes are referenced with respect to the Dart's direction rather than the parent Encoding Edge's direction.

While Encoding Edges and Darts record the GEM's relation to the original graph, Encoding Nodes strictly record the ordering of their associated edges. They more directly relate

(a) (b)

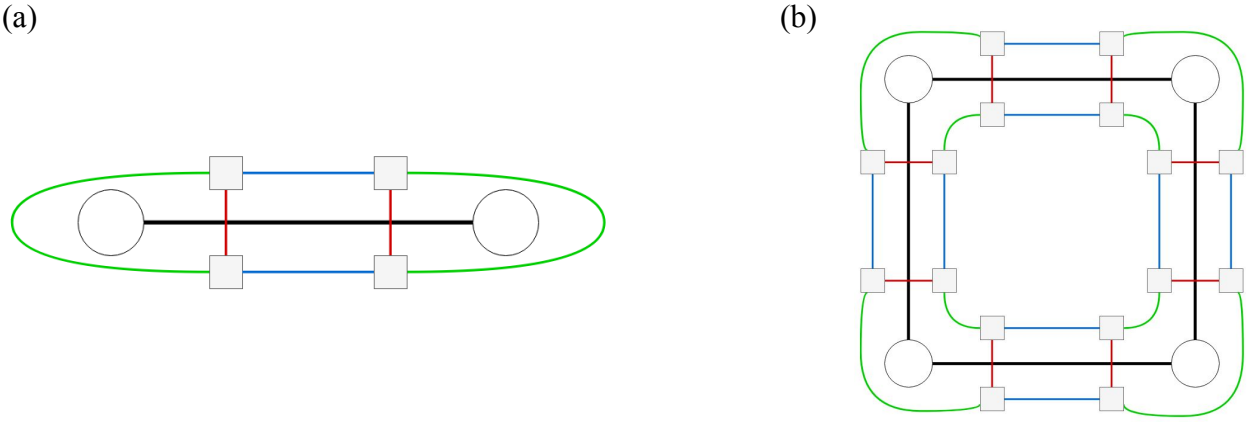


Figure 13. Basic Graph Encoded Maps. (a) A GEM representing a single edge. (b) A GEM representing a square graph. When constructing the square from individual edges, we only modify the green edges.

to the rotation system. Each Encoding Node has a reference back to its parent Encoding Edge, a reference to the vertex it is adjacent to, and a Boolean flag denoting which side of its parent edge that it is on. To implement the colored edges, we give each Encoding Node three named references to other Encoding Nodes. These are the vertex-orbit-node, face-orbit-node, and cross-node corresponding to green, blue, and red edges respectively. These references are initialized as shown in Figure 13 but can be modified later to construct arbitrary embeddings.

CHAPTER IV

GENETIC ALGORITHM IMPLEMENTATION

IV.1. Genome Representation and Processing

By definition, our genetic algorithm requires a genome to represent different graph embeddings. This genome is typically a set of series of numbers that encode various properties of a solution instance. In the case of graph embeddings, we have a candidate for this already: rotation systems.

However, rotation systems are not immediately conducive to identifying edge crossings, and it would be prohibitively difficult to enforce the rules specific to pNML during the recombination and mutation stages. Our solution to this, based on the approach from [34], is to use a genome that encodes a permutation of the edges in the graph. We pair this with a purpose-built algorithm for constructing a GEM from a permutation of edges. Together these let us identify crossings while limiting ourselves to valid embeddings without having to use a verification or filtering step.

To create a genome, we begin with the observation that all Magnets in a Magnet Layout have a unique ID, and they all have a set of outbound edges. Then the set of edges with a shared origin Magnet can be permuted. The set of Magnets itself can also be permuted. Then for a Magnet Layout with $|M|$ magnets, we can create a genome consisting of $|M| + 1$ chromosomes. The first chromosome is a permutation of the magnet IDs. For the remaining chromosomes, we first assign unique IDs for every edge and use the same ID mapping for all genomes associated

with this Magnet Layout. For Magnet m, we then define chromosome $m + 1$ as a permutation of the edges with Magnet m as their origin. Thus the first chromosome can be viewed as a permutation of the remaining $|M|$ chromosomes. We can construct a permutation of all the edges by appending the edge-chromosomes to each other in the order specified by the magnet-chromosome. This always produces a permutation where edges with the same origin are adjacent in the permutation.

To construct a GEM from an edge permutation, we observe that manipulating the vertex orbits or face orbits never changes blue or red edges. Only green edges are affected. Thus we can add edges to a GEM by inserting them into the cyclically-linked lists described by the red-green cycles, and this only requires modifying the green edges.

Construction of a GEM begins with creating an unlinked Encoding Edge for each edge in the permutation. We also create two lookup tables. The first records the leading-dart for each vertex orbit. If it exists, this is the edge that a new edge will be placed clockwise to. The other table maps each vertex to its set of connected vertices. This lets us efficiently identify vertices that belong to the same connected component.

We iterate over the Encoded Edges and run into one of five cases on each iteration, the first three being rather trivial. In Case 1, neither the origin nor the destination for the new edge have been added yet. Then none of the Encoding Nodes need modification. We record this edge's darts as the leading-dart for both vertices. We also record both vertices as belonging to the same 2-vertex connected component.

In Case 2, the origin has been added but not the destination. Then we retrieve the origin's leading-dart and add the new edge's origin-dart clockwise to it. We do this by modifying the

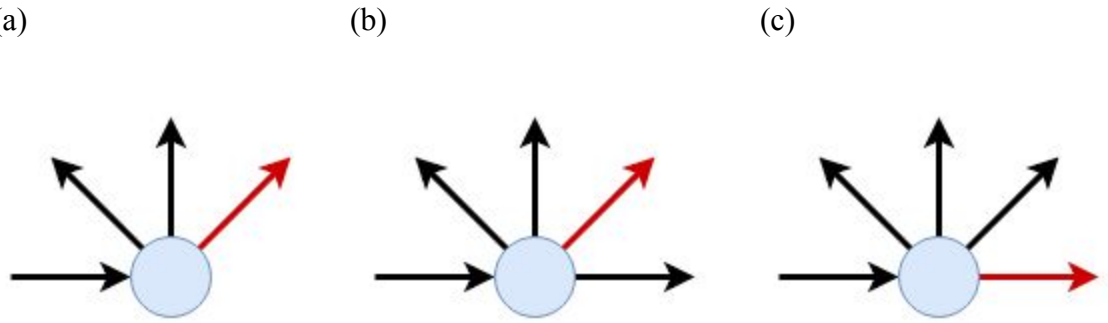


Figure 14. Adding to an Origin's Orbit. (a) The initial orbit with the leading dart in red. (b) The orbit after adding the new dart to the clockwise to the leading dart. (c) Marking the new dart as the leading dart for its origin.

green edges for the two darts. We then mark the new edge's darts as the leading-darts for the origin and destination, and we add the destination to the origin's connected component. In Case 3, we instead have already added the destination; and the origin is new. Then we follow the same procedure as in Case 2, but with the roles of the origin and destination reversed.

At this point, we pause to observe the kinds of orbits and graphs representable using only these three cases. Specifically, they are sufficient to embed trees. Inspecting these trees, we also find that the clockwise ordering of vertex orbits always matches the corresponding subsequences of edges in the original permutation. Because edges sharing the same origin are adjacent in the permutation, they are thus adjacent in the vertex orbits. This is one of the primary restrictions imposed by pNML, and we are guaranteeing its satisfaction without additional work. The remaining two cases will let us handle any graph while continuing to satisfy this requirement.

Case 4 occurs when both the origin and destination have been embedded, but they are disconnected. Then we have to pick a face in each component and merge them into one. We choose the face of the right of the leading-dart of each vertex, called the leading face. We then insert the edge to the right of both leading darts, modifying eight vertex-orbit-nodes to change

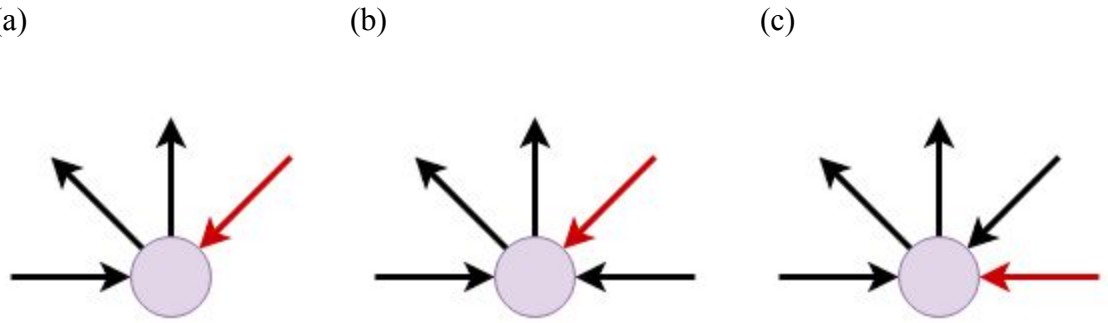


Figure 15. Adding to a Destination's Orbit. (a) The initial orbit with the leading dart in red. In this example, we have already embedded all of the destination's outbound edges. (b) The orbit after adding the new dart to the clockwise to the leading dart. (c) Marking the new dart as the leading dart for its destination.

four green edges in the GEM. We mark the new edge's darts as the leading darts for both vertices and combine the sets denoting the two connected components.

Case 5 occurs when the origin and destination are already connected. Where Case 4 can be thought of as the case that induces crossings, Case 5 is where we identify and record them. Thus we have subcases 5a and 5b, corresponding to whether we can or cannot add the edge in the plane. To check this, we first get the origin's leading face as we did in Case 4, and check if the destination is in that face. If it is, then we need to inspect its vertex orbit to make sure the new edge will not split its outbound edges. If both of these are satisfied, then we have Case 5a and can add the edge. Otherwise we have Case 5b.

In Case 5a, we add the edge as usual, updating four green GEM edges by changing eight vertex-orbit-node pointers. The vertices are already connected, so we do not need to update the connected component sets. We record the new dart as the leading dart of the origin, but we do not change the leading dart of the destination. This is to make debugging and verification simpler. In Case 5b, we record the edge as a failed edge and skip adding it.

(a) (b)

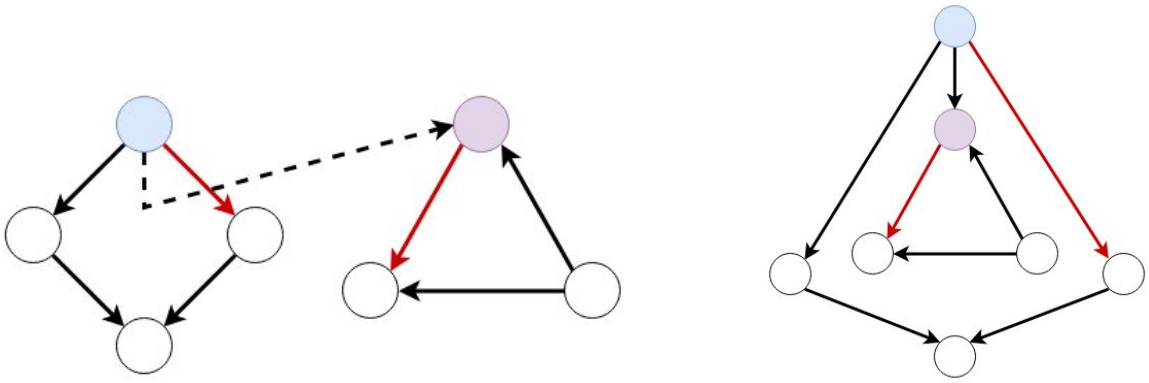


Figure 16. Joining Disconnected GEMs. (a) The origin and destination are in separate components. We will add the new edge to the clockwise to both leading darts. (b) In this example, we merged the inner face of the left component with the outer face of the right component.

We find that Cases 4 and 5 still maintain our requirements. In Case 4, we add the new edge clockwise to the leading-dart of both vertices. Thus, when isolated from Case 5, Cases 1 through 4 cannot break the ordering of outbound edges or make them nonconsecutive. For Case 5a, we add the edge clockwise to the origin's leading dart; so its outbound edge order must match the edge permutation. We also only reach Case 5a if the new edge will not split the destination's outbound edges. The destination vertex's outbound edges are either already in order or have not been added yet, so adding the new edge does not invalidate its orbit.

The end result of this algorithm is a GEM embedded on a plane with no crossings and a list of failed edges. The GEM's vertex orbits are unambiguously determined by the genome, and all embedded outbound edges are both in the same order as in the genome and are all adjacent in the embedding. We can use the number of failed edges as a cost function for the genome.

(a) (b)

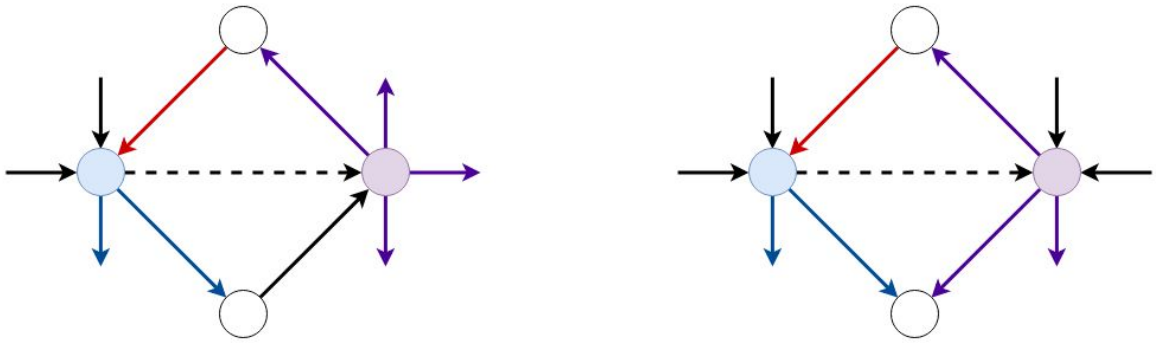


Figure 17. Identifying Disallowed Edge Embeddings. (a) The new edge is allowed because routing the edge through the leading face does not split the destination's outbound edges, shown in purple. (b) The new edge is disallowed and will be omitted. This is because the destination's outbound edges would be separated.

IV.2. Core Genetic Algorithm

Given our choice of genome and cost function, we use a standard genetic cost-minimization algorithm Each iteration begins with a population of $|P|$ genomes, with the first generation being randomly generated. We evaluate the cost of each genome and sort them in ascending order. To choose candidates for recombination, we filter the population using tournament selection. Given a selection pressure p, provided as a hyperparameter, we choose $0.5|P|$ genomes randomly with replacement. At each selection, a given genome will be chosen with probability

$$p\,(1-p)^i \tag{4}$$

where i is its index in the sorted list. The genome with the fewest failed edges has index $i = 0$; and the genome with the most has index $i = |P| - 1$. From this selected population, we choose

$|P|$ first-parents and $|P|$ second-parents independently and uniformly randomly. Each of these $|P|$ pairs will be recombined to create one new genome.

We use two recombination functions, chosen with probability 0.5 for each chromosome in each pair of parents. These functions are order-crossover (OX) and partially-matched crossover (PMX). In both cases, we randomly choose a slice of the first parent to preserve. As stated in [34] and as indicated by our own algorithm for constructing an embedding from a genome, the later edges in the permutation are more influential on crossings than the earlier ones. Thus we adapt their approach for choosing the left and right indices of this preserved slice. For the left index, we use a normal distribution centered at $\mu = 0.5|C|$ with standard deviation $\sigma = max(1,\ 0.1|C|)$ where $|C|$ is the length of the chromosome. For the right index, we instead use $\mu = |C|$. We truncate each distribution to $[0,\ |C|)$, normalize its curve to have area 1, and partition the domain into $|C|$ bins to ensure the output is a valid index.

For the OX algorithm illustrated in Figure 18, we begin with the chosen slice and copy those elements from the first parent. We then copy the leftover elements from the second parent, beginning with the first unadded element after the slice and placing it in the first unfilled space in the new chromosome. We traverse the second parent chromosome to the right, wrapping around to the first element. We add each new element encountered to the next open space in the new chromosome, wrapping around to the first slot. We stop when we have added all elements.

For the PMX algorithm illustrated in Figure 19, we also copy the first parent's selected slice. We then focus on those unassigned elements in the second parent that occupy those same indices. To resolve the collisions, we use a recursive function to assign the collided elements. To do this, we consider element *e* with index *i* in the second parent. The first parent has element *f* at

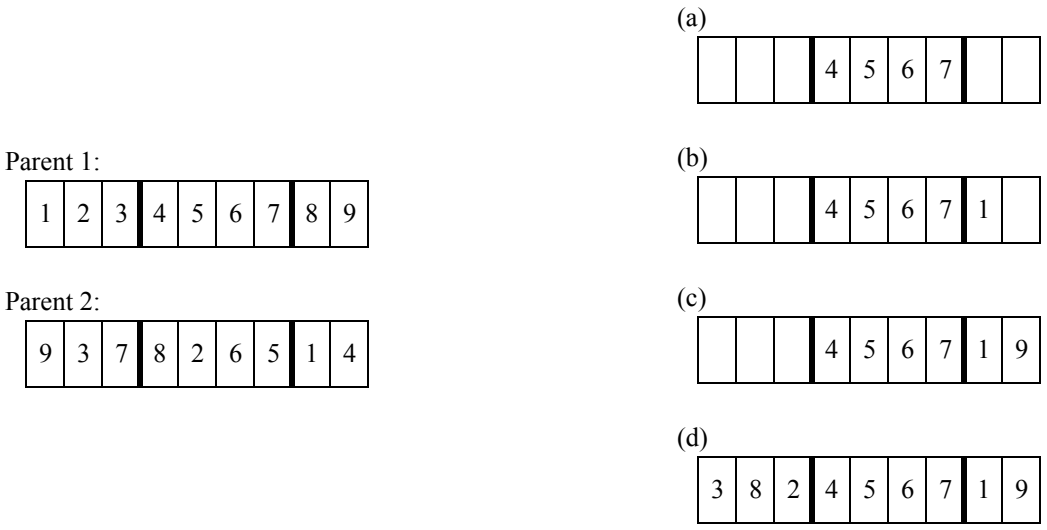


Figure 18. Order Crossover Example. (a) We begin by copying the selected slice of Parent 1. (b) We insert elements into the child chromosome, beginning with the first new element in Parent 2 after the slice. (c) We skip elements that were included in the slice. (d) We wrap around to the beginning of the child chromosome and fill the remaining locations.

index *i*, and *f* is at index *j* in the second parent. If *j* is an index in the preserved slice, then we repeat this lookup with element *g* at index *j* in the first parent. When we reach an unassigned index, we copy *e* to that location in the new chromosome. All remaining elements are then copied directly from the second parent without changing their locations.

After recombining all chromosomes in all genome pairs, we apply mutations to perturb the population. For every chromosome in the newly generated population, we choose one of four mutation functions with probability *m* or apply no mutation with probability 1 - *m*, where *m* is a hyperparameter for the probability of a mutation occurring. The four possible mutation types are an Invert, Scramble, Swap, or Insert mutation; and all four are equally probable. All four require choosing a left and right index, and we choose these according to the same distributions used by the OX and PMX algorithms. The Invert and Scramble mutations operate on the slice specified by those indices. As the names imply, the Invert mutation reverses the order of the slice; and the

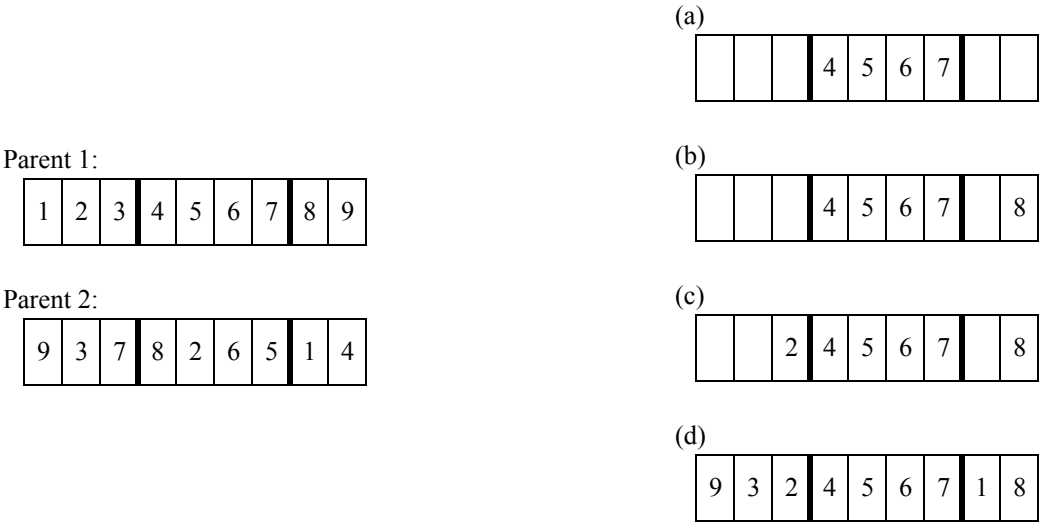


Figure 19. Partially-Matched Crossover Example. (a) We begin by copying the selected slice of Parent 1. (b) 8 is the first unmapped element in Parent 2 that is in the slice. In Parent 1, 4 occupies that location. In Parent 2, 4 is not in the slice; so we place 8 in that location. (c) We repeat this with 2, but Parent 2's copy of 5 is also in the slice. So we check 7's location and use it. (d) We copy the remaining elements of Parent 2 to the same locations.

Scramble mutation randomly shuffles it. The Swap mutation swaps the positions of the two specified elements. In our implementation, the Insert mutation takes the right element and inserts it to the left of the left element, shifting elements one space to the right to fill the gap.

Because we generally do not know the minimum possible number of failed edges or the number of generations necessary for the population to converge, we use an indirect method of deciding when to terminate the genetic algorithm. First, we require a minimum number of iterations

$$I_{min} = \lfloor \sqrt{|E|} \rfloor \; if \; |E| \geq 900, \; 30 \; otherwise \tag{5}$$

where $|E|$ is the number of edges in the circuit's graph. After I_{min} iterations, we use a smoothed rate of change in the sequence of best solutions so far. This smoothing is done using an exponential average with the smoothing parameter defined as Equation 6.

$$s = \sqrt{|E|}^{-1} \tag{6}$$

The equation that we smooth is

$$b(i) \;=\; -\frac{\Delta B(i)}{\Delta i} = B(i-1) - B(i) \tag{7}$$

where $B(i)$ is the number of failed edges in the best solution up to generation i. Because we are minimizing $B(i)$, we negate $\Delta B(i)$ to ensure that $b(i)$ is positive when the algorithm finds better solutions. This makes the termination logic and user interface more intuitive. Equation 8 is the final recursive equation that we monitor after I_{min} iterations.

$$T(i) = s\, b(i) + (1-s)\, T(i-1) \, for \; i > 0; \;\; T(0) = 0 \tag{8}$$

The algorithm terminates when $T(i)$ falls below a user-defined threshold and after at least I_{min} iterations. Figure 31 in Appendix A plots $T(i)$ for an experiment using ISCAS circuit c6288, alongside the algorithm's performance for the test case.

IV.3. Spark Implementation

One of the biggest advantages of genetic algorithms is that they are extremely parallelizable. Each genome fitness or cost is an independent calculation, as is the recombination of every pair of selected chromosomes, and every chromosome mutation. We opted to take advantage of this by implementing our algorithm using Spark. To fit the table-like model of its Datasets, we slightly reorganized the standard genetic algorithm and fragmented genomes to improve parallelization. Each iteration works with a pipeline comprised of eight primary Datasets: the magnets, edges, population, scored population, selected population, parent pairings, new population, and new scored population.

The magnets and edges Datasets are foundational lookup tables to ensure we always have a well-defined link back to the original Magnet Layout. The magnets Dataset records the list of magnet IDs alongside how many outbound edges each has. The edges Dataset records an enumeration of the origin-destination magnet ID pairs in the Magnet Layout's adjacency list. Both of these are invariant throughout the evolutionary algorithm execution and are used to create the initial population.

To create the initial population Dataset, we first create a population RDD consisting of one genome. To maximize parallelization, each row in the RDD will be a single chromosome. Thus the key that identifies a chromosome is a genome ID paired with a chromosome ID. We compute the first chromosome by adding the chromosome ID column to the magnets Dataset, with all chromosome IDs being 0. Similarly, we add a chromosome ID column to the edges Dataset, except the ID given to an edge is its origin magnet ID, plus one. We select just the chromosome and manget/edge ID columns from both tables and take the union of the resulting two-column RDDs. This produces a single RDD of chromosome-element ID pairs. To produce the first genome, we group all of these pairs by chromosome ID, turn each group into a chromosome ID paired with a list of element IDs, and add a constant-0 genome ID column. To turn this into a complete population of size $|P|$, we map each chromosome to $|P|$ randomly-shuffled copies of itself, enumerate the resulting RDDs, and take the union of them all. The result is an RDD with $|P|$ uniquely-keyed permutations of every chromosome, and we convert this into a Dataset. This is shown in Figure 20.

Before beginning the evolutionary process, we use our cost function to score the initial population. This modification to the standard genetic algorithm makes it easier to monitor the

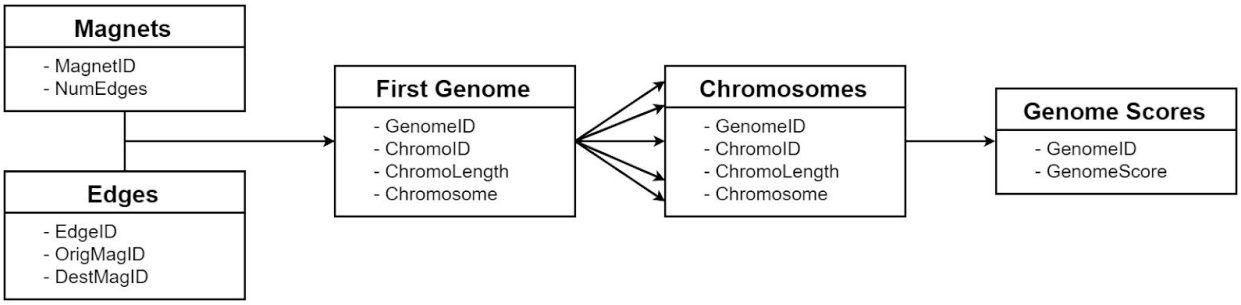
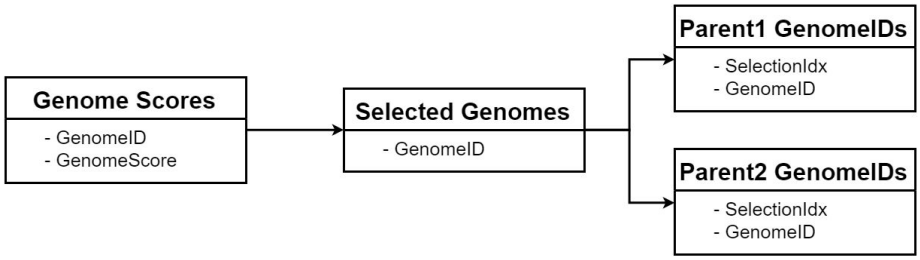


Figure 20. Creation and Scoring of a Spark Genome Population. We create the initial population by transforming the Magnets and Edges Datasets, joining them into a single genome, and reshuffling that genome. We then assign scores to each genome by processing the chromosomes.

algorithm's performance without putting monitoring functionality directly in the algorithm itself. To use our cost function, we simply group the population's chromosomes by their genome IDs. We then aggregate the groups into mappings between chromosome IDs and chromosomes. This representation is isomorphic to our discussed 2D-array representation of a genome and can be converted into a permutation of edge IDs using the associated algorithm. We then use the edges Dataset to map edge IDs to pairs of magnet IDs and apply our cost function to the permutation of origin-destination paris. This results in $|P|$ independent calculations distributed automatically across the Spark cluster. The final result is a Dataset where each element is a genome ID paired with the number of failed edges.

Each iteration of the genetic algorithm begins with the magnets, edges, population, and scored population Datasets. The algorithm does not make use of the magnets Dataset, but we keep it for verification purposes as it contributes negligibly to the total memory costs of the algorithm. To perform a single iteration, we perform tournament selection on the population, randomly select parent pairings, recombine parents using crossover functions, mutate the resulting genomes to produce the next population, and evaluate the costs for the new population.

(a)



(b)

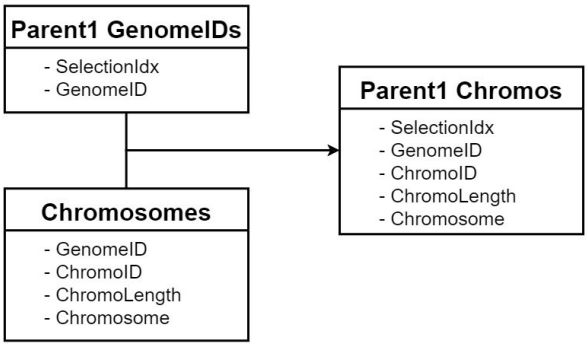


Figure 21. Selection of Parent Genomes. (a) We use tournament selection to sample from the current population. We then sample from that selected population to choose two sets of parent genome IDs. (b) We use the parent genome IDs to filter the population's chromosomes to obtain the parent chromosomes.

To perform tournament selection we sort the scored population Dataset by the cost column convert to an RDD, and enumerate the rows according to their order with the best genome having index 0 and the worst having index $|P|-1$. We then map this index i to the tournament selection probability $p\,(1-p)^i$ where p is the given selection pressure. From this RDD, we select $0.5|P|$ genomes using distributed weighted random sampling with replacement. The result is an RDD comprised of $0.5|P|$ genome IDs, and Figure 21a illustrates the pipeline.

We need two such RDDs of length $|P|$ to create our parent pairings, and we can do this by using distributed uniform sampling with replacement on the selected population. We

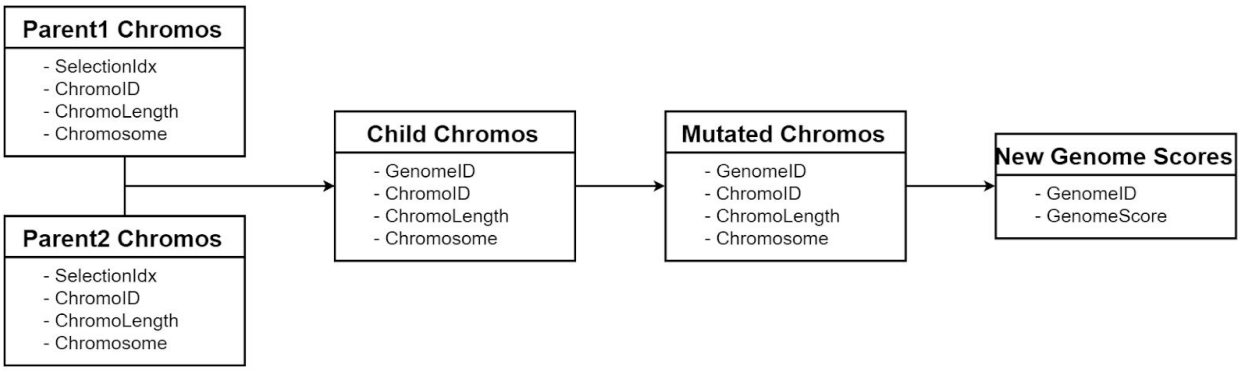


Figure 22. Recombination and Mutation of Child Genomes. We join the parent chromosome Datasets and use recombination to produce the Dataset of child chromosomes. We then mutate these chromosomes to obtain the next population and score it as in Figure 20.

enumerate both sets of samples and join them back to the population Dataset to retrieve the selected chromosomes, as shown in Figure 21b. This gives us two Datasets where each row is a selection index, a genome ID, a chromosome ID, and a chromosome. We can then join these together by joining on the selection index and chromosome ID columns.

Because recombination and mutation are per-chromosome operations, our chromosome pairings are in a Dataset that lets Spark maximally parallelize generation of the next population. To do this, we simply transform each pairing using a function that randomly picks and applies a crossover function to generate a child genome. We then transform each child genome using a function that randomly picks and applies a mutation function. Thus each pair of parents from the current population is mapped to mutated child genome in the next population, and each child can use the parent-pair's selection index as its genome ID. We then score this new population using the same algorithm used to score the initial population, and we have completed a single iteration of the distributed genetic algorithm. We can return the new population, new scored population, magnets, and edges Datasets for analysis and use in the next iteration.

IV.4. Genetic Algorithm Results

For our genetic algorithm, we have to set four hyperparameters. These are the stop rate, population size, selection pressure, and mutation probability. We set these to the values shown in Table 2. Table 3 shows the final results, and Figures 23-32 in Appendix A show per-generation results for each experiment where the first plot shows scoring statistics for each generation, and the second shows the value of the termination function at each generation. All circuits are combinational circuits taken from the ISCAS Verilog benchmarks. We ran these on an Amazon EC2 c5d.4xlarge instance with 12 cores and 25 GiB of RAM allocated to Spark.

Table 2: Hyperparameters Set for Genetic Algorithm

Parameter Name	**Symbol**	**Value**
Stop Rate	r	0.1
Population Size	$\|P\|$	100
Selection Pressure	p	0.2
Mutation Probability	m	0.3

Table 3: Final Results for Genetic Algorithm

Test Name	**Num Edges**	**Num Iterations**	**Exec Time**	**Initial Num Crossings**	**Final Num Crossings**	**Crossing Reduction**
c17	24	31	00:01:12	0	0	0%
c432	892	76	00:06:28	140	119	15.00%
c880	1619	88	00:10:20	207	184	11.11%
c499	1754	96	00:10:32	302	271	10.26%
c1355	2202	119	00:16:13	270	235	12.96%
c1908	2970	131	00:22:24	383	348	9.14%
c2670	4455	136	00:33:11	478	444	7.11%
c3540	6371	161	00:56:04	946	893	5.60%
c6288	9824	292	02:24:43	1390	1298	6.62%
c5315	10142	192	01:48:50	1442	1384	4.02%

CHAPTER V

REINFORCEMENT LEARNING IMPLEMENTATION

V.1. State and Action Representation

For our reinforcement learning algorithm, we keep the same combinatorial approach as we used for the genetic algorithm. Recall that reinforcement learning algorithms model their problems as Markov Decision Processes (MDPs), paired with a learning actor. Both of these have to process system states and actions applied to those states. Thus we begin by designing the representation of states and actions.

We first consider the kinds of operations we want to permit on a graph embedding. Given our understanding of GEMs, we choose to permit perturbation of either a vertex orbit or the ordering of those orbits. To reuse terminology from our genetic algorithm, we will let the actor modify one of the chromosomes by moving an element to the right by one position. If the agent chooses the rightmost element, that element will be moved to the leftmost position. Thus, if we reuse the same genome representation of a permutation of edges, an action is a pair of indices denoting the chromosome to modify and the element to move. The result is a deterministic MDP that accepts a 2D array and index pair and outputs a 2D array with one row modified as we have described.

We also research the cost function from our genetic algorithm because we use the same representation for graph embeddings in both approaches. To calculate the reward for an action,

we simply pass the state to our construction algorithm, count the number of failed edges, and return the negative of the square of that count. Returning a negative reward on each step will encourage the actor to minimize crossings quickly, and using the square of the number of failed edges more strongly emphasizes the cost of bad solutions.

V.2. State and Action Encoding

Because we plan to use a neural network to choose actions based on system states, we have to encode system states and actions in a way that is suitable for a neural network. This means our input signals should be in the range [-1, 1], and our actions have to be indexed. We also need these to be as compact as possible, so the network is smaller.

We do this by recognizing that not every chromosome can be modified. Many of them are empty or only have one element. In these cases, we do not need to implement any actions. Those chromosomes also are invariant across states, so there is nothing for the neural network to learn from them. We can omit these chromosomes from the vectorized state. Thus, given the list of chromosomes, we will first filter out those that have fewer than two elements.

To ensure our states use signals in the range [-1, 1], we use a one-hot encoding. To do this, we treat each element's location as a bin and tell that bin which element it contains using a one-hot encoding of the element. A naive encoding might use one-hot vector of length equal to the number of elements in the filtered genome, but this would waste a huge amount of space. A given bin will only ever hold one of the elements of its associated chromosome, so the vectors it receives should be the same length as that chromosome. To pick the exact encoding for an element, we sort its chromosome and assign it to its index in the resulting sorted list. This index

will be the index of the 1 in its one-hot vector. To create the final state vector, we first vectorize the chromosomes by mapping each element to its one-hot vector and concatenating those vectors in the order specified by the chromosome. We then concatenate the vectorized chromosomes in order to create a single binary vector.

To encode our actions, we recall that each action moves the element at a location to the right. This corresponds to the index of one of the bins. This indexing can be defined by concatenating the non-vectorized chromosomes to create a single vector of decimals. This implies a bijection between chromosome-element index pairs and vector indices.

Decoding these indices to get a chromosome-element index pair, an unencoded action, is somewhat tricky. If every chromosome was the same length l, we could just use $f(i) = (\lfloor i / l \rfloor, i \% l)$ where % denotes a modulus. However our chromosomes are of varying lengths. Let C' be the set of chromosomes with two or more elements and let them be ordered so that c_k is the kth chromosome to be encoded. If chromosome c_k has length l_k, then we use Equation 8 to compute the base offset for a chromosome.

$$o_k = o(c_k) = \sum_{i=0}^{k-1} l_k \tag{8}$$

Then we create lookup tables $M_{co}(k) = o_k$ and $M_{oc}(i) = k_i$ where k_i is the the index of the chromosome containing the ith bin, which we can use to identify chromosome c_{k_i}. Then given an action encoded as the scalar value A_s, we decode it with Equation 9.

$$A = (M_{oc}(A_s),\ A_s - M_{oc}(A_s)) \tag{9}$$

To encode action $A = (k,\ b)$ as a scalar, we use Equation 10.

$$A_s = M_{co}(k) + b \tag{10}$$

V.3. Neural Network Topology

These encoding schemes determine the dimensions of our neural networks input and output layers. The input vector has length $\sum_{c \in C'} l_c^2$ where C' is the set of chromosomes with two or more elements, and l_c is the length of chromosome c. Thus we need that many nodes in the input layer. For the output layer, we note that we will implement an action-value function; so we need a node for each allowed action. This gives us $\sum_{c \in C'} l_c$ output nodes.

The remainder of the topology is largely up to the engineer. The network will be learning a regression, so we opted to use Identity activations on the output layer. All other layers use the Leaky ReLU activation function with a slope of 0.01 for negative input signals. We only used one deep layers of length $\sum_{c \in C'} l_c$ which is the same size as the output layer. All layers were fully connected to their neighboring layers. We initialized the weights using Xavier initialization and used the Nesterov's momentum to update them. The loss function applied to the output signal was Mean Squared Error. We implemented all of this using the DeepLearning4J library [41].

V.4. Training and Execution

We used Q-Learning with experience replay as our core learning procedure. Recall that at every step, we have the current state S. Given S and our current value function Q, our policy $\pi(S)$ returns an action A with expected scalar reward $Q(S, A_s)$. We implement Q using the neural network from the previous section, and we use an epsilon-greedy policy for choosing A. Given

the vector output by $Q(S)$, π returns a random action with probability ε. Otherwise it returns the action with the highest predicted total return. This chosen action A_s is the scalar representation of the action and must be decoded to get the usable representation A. We send (S, A) to the environment to receive the reward R and the next state S'. The result is a tuple (S, A, R, S') which we call a Replay. We save this tuple to a buffer for use in offline training.

For each episode, we run $2|E|$ steps where $|E|$ is the number of edges in the graph. At the end of each episode, we sample 20% of the replays from the buffer without replacement and train on them. Given the replay (S, A, R, S'), we want to update our predicted value $Q(S, A_s)$. To do this, we first compute the next action A' as $A' = \pi(S')$. Our new estimate for R is then given by

$$Q^{+}(S,\ A_s)\ =\ Q(S,\ A_s) + \alpha(R + Q(S',\ A_s') - Q(S,\ A_s)) \tag{11}$$

where α is the learning rate. This is the same Q-Learning update function as Equation 1, except we use a discount of $\gamma = 1$ here. Then our training vector's elements are $Q^{+}(S,\ A_s^{*})$ for $A_s^{*} = A_s$ and $Q(S,\ A_s)$ otherwise. We update the neural net using this training vector and the Nesterov's momentum update algorithm with the Mean Squared Error.

We use the same type of termination algorithm as we did for the genetic algorithm. That is, we track the total reward at the end of each episode and are interested in episodes that yield a better global maximum. We stop running episodes when the exponential average of the current-best value drops below a given threshold, with a minimum of thirty episodes.

V.5. Reinforcement Learning Results

Our reinforcement learning algorithm has 3 hyperparameters: The stop rate, the learning rate, and the policy's exploration probability. We set these to the values shown in Table 4. We only ran tests for circuit c432 as this was the smallest non-trivial test case. Its twelve-hour execution time was sufficient to forego further testing. The other benchmark circuits would have taken far longer to train and would have learned even more slowly. Table 5 shows the final results. The final number of crossings is determined as the number of crossings output by the last episode because this was the episode with the most-trained network.

Table 4: Hyperparameters Set for Reinforcement Learning Algorithm

Parameter Name	**Symbol**	**Value**
Stop Rate	r	0.1
Learning Rate	α	0.2
Exploration Probability	ε	0.2

Table 5: Final Results for Reinforcement Learning Algorithm

Test Name	**Num Edges**	**Num Iterations**	**Exec Time**	**Initial Num Crossings**	**Final Num Crossings**	**Crossing Reduction**
c432	892	30	12:12:07	132	694	-425.76%

CHAPTER VI

RANDOM SEARCH IMPLEMENTATION

VI.1. Adaptation of Genetic Algorithm

A significant limitation of the ISCAS benchmarks is that we do not know their minimum number of crossings possible under pNML's constraints because the problem is NP-Complete; and pNML's variation of it is effectively unexplored. In the absence of a target lower-bound, we need control cases that our genetic and reinforcement learning algorithms can try to improve upon. The natural candidate for this, given that the problem is NP-Complete, is the random search. If our algorithms perform similarly to a random search, then we can conclude that they were unsuccessful without knowing the optimal solutions. We implemented the random search by slightly modifying the genetic algorithm. Instead of iteratively applying the selection, recombination, and mutation algorithms, we generate a new "initial" population from scratch according to a uniform distribution. This makes the algorithm useful as a blind-search baseline as it only incorporates the knowledge necessary to produce valid embeddings.

We used two different termination conditions. The first was the exact same condition as the previous two algorithms, based on the rate of improvement. The second set a maximum number of iterations. We use the latter to test the scenario where the random search generated the same number of samples as the genetic algorithm, so we can better compare execution times.

VI.2. Random Search Results

Both experiments with the random search had the population size as a hyperparameter. The experiment where the termination condition was a minimum rate of improvement had that minimum rate as a hyperparameter as well. The experiment that instead used a maximum number of iterations had that number as a hyperparameter instead.

Both experiments used settings that matched the genetic algorithm's experiment detailed in Tables 2 and 3. The first experiment used a population size of 100 and a stop-rate of 0.1. The second experiment also used a population size of 100. The maximum number of iterations for each test case was set to match the number used by the corresponding test case for the genetic algorithm, given by the Num Iterations column in Table 3. Table 6 and Figures 33-42 in Appendix B show the final results for the ISCAS benchmarks using the minimum rate of improvement termination condition. Table 7 and Figures 43-52 in Appendix C show the results while using the maximum number of iterations termination condition. The figures in Appendix C include plots of $T(n)$ for comparison against the figures in Appendices A and B.

Table 6: Final Results for Random Search Algorithm with Stop Rate

Test Name	Num Edges	Num Iterations	Exec Time	Initial Num Crossings	Final Num Crossings	Crossing Reduction
c17	24	31	00:00:41	0	0	0%
c432	892	31	00:01:31	139	137	1.44%
c880	1619	41	00:03:06	207	199	3.86%
c499	1754	55	00:03:48	304	295	2.96%
c1355	2202	47	00:04:20	273	266	2.56%
c1908	2970	66	00:07:55	383	367	4.18%
c2670	4455	67	00:12:18	481	473	1.66%
c3540	6371	80	00:22:12	946	932	1.48%
c6288	9824	100	00:40:40	1393	1377	1.15%
c5315	10142	123	01:00:07	1447	1422	1.73%

Table 7: Final Results for Random Search Algorithm with Max Iterations

Test Name	Num Edges	Num Iterations	Exec Time	Initial Num Crossings	Final Num Crossings	Crossing Reduction
c17	24	31	00:00:42	0	0	0%
c432	892	76	00:03:18	145	135	6.90%
c880	1619	88	00:05:28	201	197	1.99%
c499	1754	96	00:05:27	300	288	4.00%
c1355	2202	119	00:08:43	273	263	3.66%
c1908	2970	131	00:12:22	377	371	1.59%
c2670	4455	136	00:18:54	478	462	3.35%
c3540	6371	161	00:38:44	936	929	0.75%
c6288	9824	292	01:32:41	1390	1374	1.15%
c5315	10142	192	01:16:43	1442	1423	1.32%

CHAPTER VII

CONCLUSIONS AND FUTURE WORK

VII.1. Conclusions and Analysis

Our genetic algorithm shows promise. We see that it consistently performed significantly better than a random NP search. Even more importantly, we see the population's average cost fluctuating with the population's lowest cost. The downward slopes shown early in each of the test cases illustrated in Appendix A demonstrates that the population evolved and improved as good substructures propagated via selection and recombination. This indicates that the problem has a learnable substructure, despite its unusual constraints.

Based on our analysis of the simulation results, we conclude that deep reinforcement learning is not the ideal solution for this problem. Our single test case for this algorithm required more execution time than all of the other presented test cases combined, but the actor barely began to explore the available actions. This was because the system state's size was linear in terms of the number of magnets $|M|$, so the vectorized state fed into the neural network was quadratic in length. This resulted in a three-layer neural net that had $O(|M|^4)$ weights to adjust, making the algorithm far too intensive to be practical. We also recognize that, if the training algorithm had worked, the resulting AI would have only been able to optimize the circuit it trained on. This is because each circuit corresponds to a different MDP, featuring different allowed states, different available actions, and a different distribution of rewards. Although different circuits are different instances of the pNML wire crossing problem, reinforcement

learning models them in a way that makes each a distinct system to learn. Designing an AI that can simultaneously learn multiple systems and generalize knowledge gleaned from each of them is an open problem in machine learning.

VII.2. Future Work

Our convergence results for the genetic algorithm indicate that the problem has a learnable substructure. We can explore this in the future by using different recombination algorithms or a more sophisticated selection algorithm. The primary challenge here is in making the algorithm more robust against local minima. In our tests, the population responded almost immediately to improvements in the best genomes; and this likely limited our algorithm's performance.

There is still limited potential for deep reinforcement learning in solving this problem. A state representation based on a spatial embedding could be far more compact than our combinatorial approach without introducing much nonlinearity. This representation would likely be a series of coordinate pairs for each vertex, and the rotation system could be recovered from these coordinates using trigonometry. Such a representation would likely require a policy gradient method of reinforcement learning, which is distinct from the value function class of methods we based our approach on. Policy gradient methods are notable for their ability to handle both continuous- and discrete-valued actions. However an AI trained with this approach would still be unable to learn to optimize multiple different circuits without significant advances in machine learning.

The pNML wire crossing problem is also only a small component of a complete EDA workflow for pNML, and it interacts with the other place-and-route subproblems. Development of just one design rule set for pNML would be a highly valuable contribution to this workflow as this is a prerequisite to several optimizations, especially for minimizing circuit area and maximizing clocking frequency. This would also permit holistic approaches that optimize multiple characteristics simultaneously.

BIBLIOGRAPHY

[1] C. S. Lent, B. Isaksen, "Clocked molecular quantum-dot cellular automata," in *IEEE Transactions on Electron Devices*, vol. 50, no. 9, pp. 1890-1896, Sept. 2003.
[2] E. Varga et al., "Non-volatile and reprogrammable MQCA-based majority gates," *2009 Device Research Conference*, University Park, PA, 2009, pp. 1-2.
[3] M. T. Niemier et al., "Nanomagnet logic: progress toward system-level integration," *J. Phys.: Condense. Matter*, vol. 23, no. 23, Nov. 2011.
[4] R. Perricone, X. S. Hu, J. Nahas, M. Niemier, "Design of 3D nanomagnetic logic circuits: A full-adder case study," *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Dresden, 2014, pp. 1-6.R. Perricone, X.
[5] C.S. Lent, G. L. Snider, "The Development of Quantum-Dot Cellular Automata," *Field Coupled Nanocomputing*, 2014.C.S. Lent, G. L. Snider, "The Development of Quantum-Dot Cellular Automata," *Field Coupled Nanocomputing*, 2014.
[6] C. S. Lent, P. D. Tougaw, W. Porod, "A bistable quantum cell for cellular automata," in *International Workshop Computational Electronics*, pp. 163–166, 1992.
[7] C. S. Lent, P. D. Tougaw, "A device architecture for computing with quantum dots," in *Proceedings of the IEEE*, vol. 85, no. 4, pp. 541-557, April 1997.
[8] V. Vankamamidi, M. Ottavi, F. Lombardi, "Clocking and Cell Placement for QCA," *2006 Sixth IEEE Conference on Nanotechnology*, Cincinnati, OH, USA, 2006, pp. 343-346.
[9] R. A. Wolkow, et al., "Silicon Atomic Quantum Dots Enable Beyond-CMOS Electronics," *Field Coupled Nanocomputing*, 2014.
[10] I. Palit, X. S. Hu, J. Nahas, M. Niemier, "Systematic design of Nanomagnet Logic circuits," *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Grenoble, France, 2013, pp. 1795-1800.
[11] R. Perricone, Y. Zhu, K. M. Sanders, X. S. Hu, M. Niemier, "Towards systematic design of 3D pNML layouts," *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, Grenoble, 2015, pp. 1539-1542.
[12] A. Orlov, et al., "Magnetic Quantum-Dot Cellular Automata: Recent Developments and Prospects," *J. Nanoelectronics and Optoelectronics*, 2008.
[13] D. Giri, M. Vacca, G. Causapruno, W. Rao, M. Graziano, M. Zamboni, "A standard cell approach for MagnetoElastic NML circuits," *2014 IEEE/ACM International Symposium on Nanoscale Architectures (NANOARCH)*, Paris, 2014, pp. 65-70.
[14] W. Liu, et al., "Security Issues in QCA Circuit Design - Power Analysis Attacks," *Field Coupled Nanocomputing*, 2014.
[15] O.Donzelli, et al., "Perpendicular Magnetic Anisotropy and Stripe Domains in Ultrathin Co/Au Sputtered Multilayers," *J. Appl. Phys.*, 2003.
[16] X. Ju et al., "Nanomagnet Logic from Partially Irradiated Co/Pt Nanomagnets," in *IEEE Transactions on Nanotechnology*, vol. 11, no. 1, pp. 97-104, Jan. 2012.
[17] W. Kaiser, M. Kiechle, G. Žiemys, D. Schmitt-Landsiedel, S. Breitkreutz-von Gamm, "Engineering the Switching Behavior of Nanomagnets for Logic Computation Using 3-D Modeling and Simulation," in *IEEE Transactions on Magnetics*, vol. 53, no. 6, pp. 1-4, June 2017.
[18] S. Breitkreutz et al., "Experimental Demonstration of a 1-Bit Full Adder in Perpendicular Nanomagnetic Logic," in *IEEE Transactions on Magnetics*, vol. 49, no. 7, pp. 4464-4467, July 2013.

[19] R. Zhang, P. Gupta, N. K. Jha, "Majority and Minority Network Synthesis With Application to QCA-, SET-, and TPL-Based Nanotechnologies," in IE*EE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 7, pp. 1233-1245, July 2007.
[20] S. Rai, "Majority Gate Based Design for Combinational Quantum Cellular Automata (QCA) Circuits," *2008 40th Southeastern Symposium on System Theory (SSST)*, New Orleans, LA, 2008, pp. 222-224.
[21] P. Wang, M. Y. Niamat, S. R. Vemuru, M. Alam, T. Killian, "Synthesis of Majority/Minority Logic Networks," in *IEEE Transactions on Nanotechnology*, vol. 14, no. 3, pp. 473-483, May 2015.
[22] M. R. Bonyadi, S. M. R. Azghadi, N. M. Rad, K. Navi, E. Afjei, "Logic Optimization for Majority Gate-Based Nanoelectronic Circuits Based on Genetic Algorithm," *2007 International Conference on Electrical Engineering*, Lahore, 2007, pp. 1-5.
[23] L. Amarú, P. E. Gaillardon, G. De Micheli, "BDS-MAJ: A BDD-based logic synthesis tool exploiting majority logic decomposition," 2*013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, Austin, TX, 2013, pp. 1-6.
[24] L. Amarú, P. E. Gaillardon, G. De Micheli, "Majority-Inverter Graph: A novel data-structure and algorithms for efficient logic optimization," *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, San Francisco, CA, 2014, pp. 1-6.
[25] M. J. Donahue et al., “Oommf user’s guide, version 1.0,” National Institute of Standards and Technology, Gaithersburg, MD, USA, Tech. Rep. NISTIR 6376, September 1999.
[26] F. Riente, U. Garlando, G. Turvani, M. Vacca, M. Ruo Roch, M. Graziano, "MagCAD: Tool for the Design of 3-D Magnetic Circuits," in *IEEE Journal on Exploratory Solid-State Computational Devices and Circuits*, vol. 3, pp. 65-73, Dec. 2017.
[27] M. Mitchell, “An Introduction to Genetic Algorithms,” 5th ed. Cambridge, MA: MIT Press, 1999.
[28] R. S. Sutton, A. G. Barto, “Introduction to reinforcement learning,” Cambridge, Mass, MA: MIT Press, 1998.
[29] S. K. Lando, A. K. Zvonkin, “Graphs on surfaces and their applications,” Berlin: Springer, 2004.
[30] J. L. Gross, S. R. Alpert, “The topological theory of current graphs,” *Journal of Combinatorial Theory*, Series B, vol. 17, no. 3, pp. 218–233, 1974.
[31] “RDD Programming Guide,” Apache Spark™ - *Unified Analytics Engine for Big Data*. [Online]. Available: https://spark.apache.org/docs/2.3.1/rdd-programming-guide.html. [Accessed: 16-Nov-2018].
[32] F. Brglez, H. Fujiwara, "A Neutral Netlist of 10 Combinational Benchmark Circuits", *Proc. IEEE Int'l Symp. Circuits and Systems*, pp. 695-698, 1985.
[33] B. Marshall, "Verilog Parser," GitHub, MIT License. https://github.com/ben-marshall/verilog-parser.
[34] B. M. Neta, G. H. Araújo, F. G. Guimarães, R. C. Mesquita, P. Y. Ekel, “A fuzzy genetic algorithm for automatic orthogonal graph drawing,” *Applied Soft Computing*, vol. 12, no. 4, pp. 1379–1389, 2012.
[35] J. Timler, C. S. Lent, “Power gain and dissipation in quantum-dot cellular automata,” *Journal of Applied Physics*, vol. 91, no. 2, pp. 823–831, 2002.

[36] M. Niemier et al., "Clocking structures and power analysis for nanomagnet-based logic devices," *Proceedings of the 2007 international symposium on Low power electronics and design (ISLPED '07)*, Portland, OR, 2007, pp. 26-31.
[37] W. j. Chung, B. Smith, S. k. Lim, "Node duplication and routing algorithms for quantum-dot cellular automata circuits," in *IEE Proceedings - Circuits, Devices and Systems*, vol. 153, no. 5, pp. 497-505, October 2006.
[38] B. Sen, A. Sengupta, M. Dalui, B. K. Sikdar, "Design of universal logic gate targeting minimum wire-crossings in QCA logic circuit," 2010 53rd IEEE International Midwest Symposium on Circuits and Systems, Seattle, WA, 2010, pp. 1181-1184.
[39] R. K. Nath, B. Sen, B. K. Sikdar, "Optimal synthesis of QCA logic circuit eliminating wire-crossings," in *IET Circuits, Devices & Systems*, vol. 11, no. 3, pp. 201-208, 5 2017.
[40] W. J. Chung, B. Smith, S. K. Lim, "QCA physical design with crossing minimization," *5th IEEE Conference on Nanotechnology*, 2005., 2005, pp. 108-111 vol. 1.
[41] Eclipse Deeplearning4j Development Team. “Deeplearning4j: Open-source distributed deep learning for the JVM,” Apache Software Foundation License 2.0. http://deeplearning4j.org.

LIST OF APPENDICES

APPENDIX A: TESTS FOR GENETIC ALGORITHM

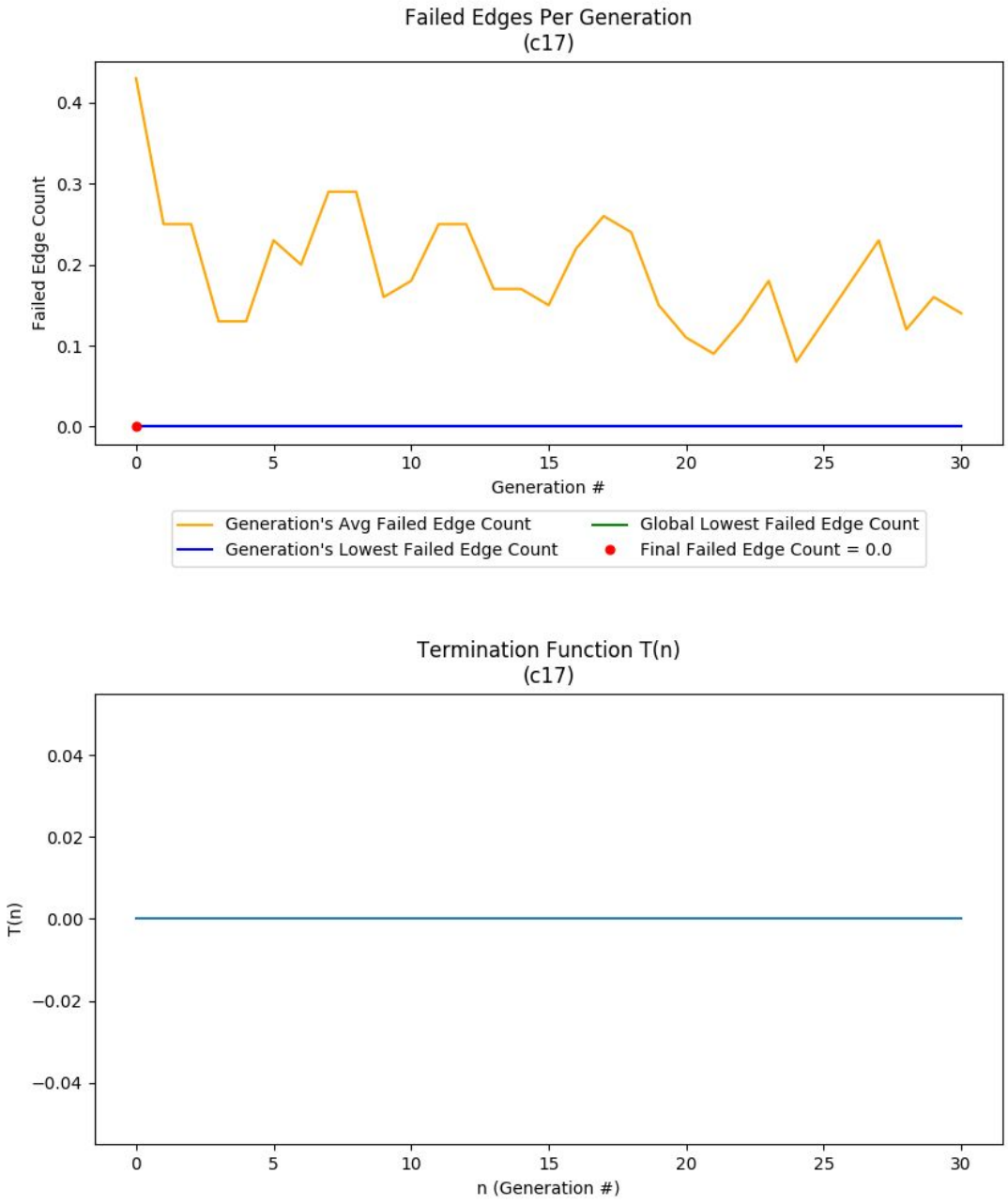


Figure 23. Genetic Results for ISCAS Test c17.

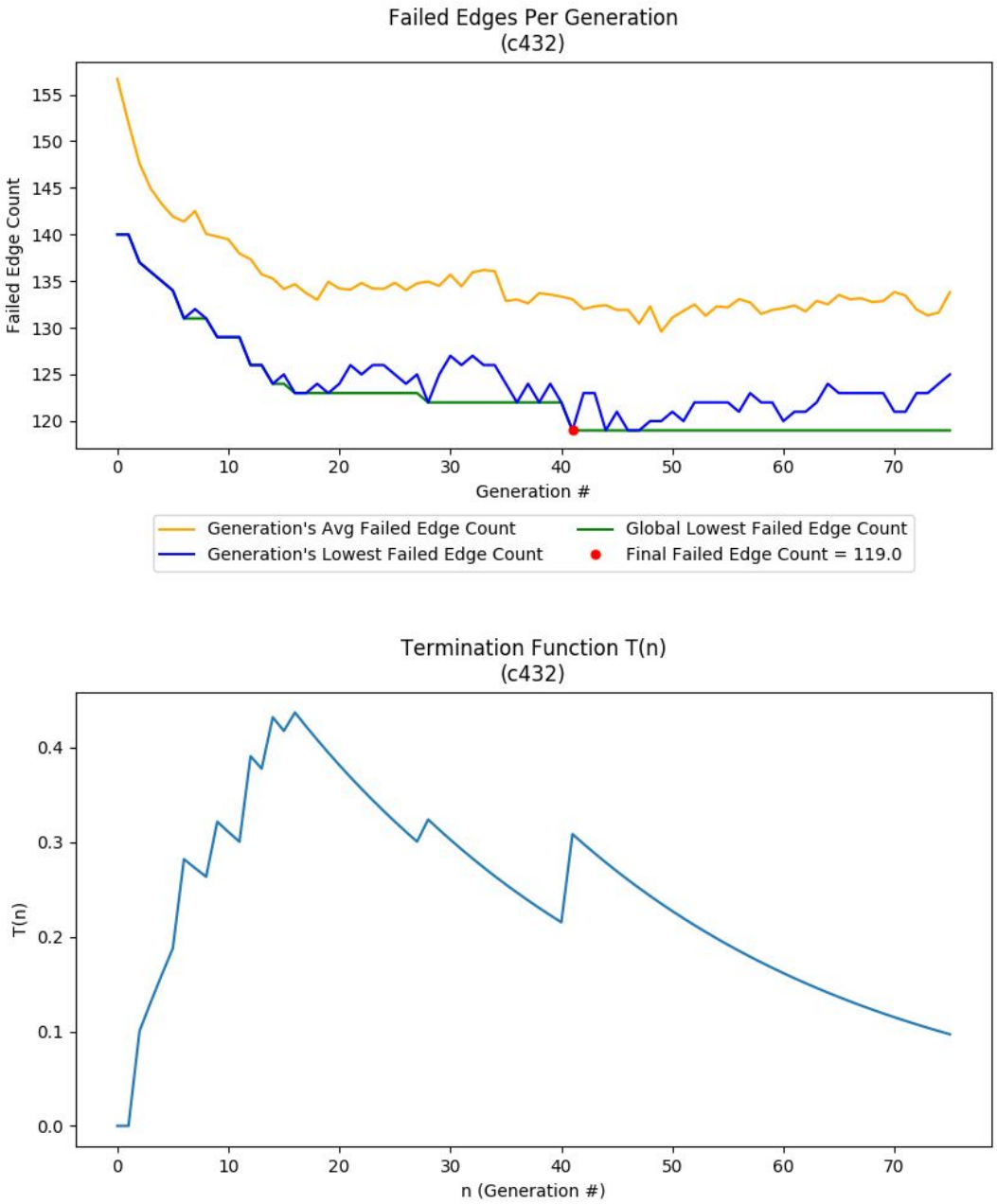


Figure 24. Genetic Results for ISCAS Test c432.

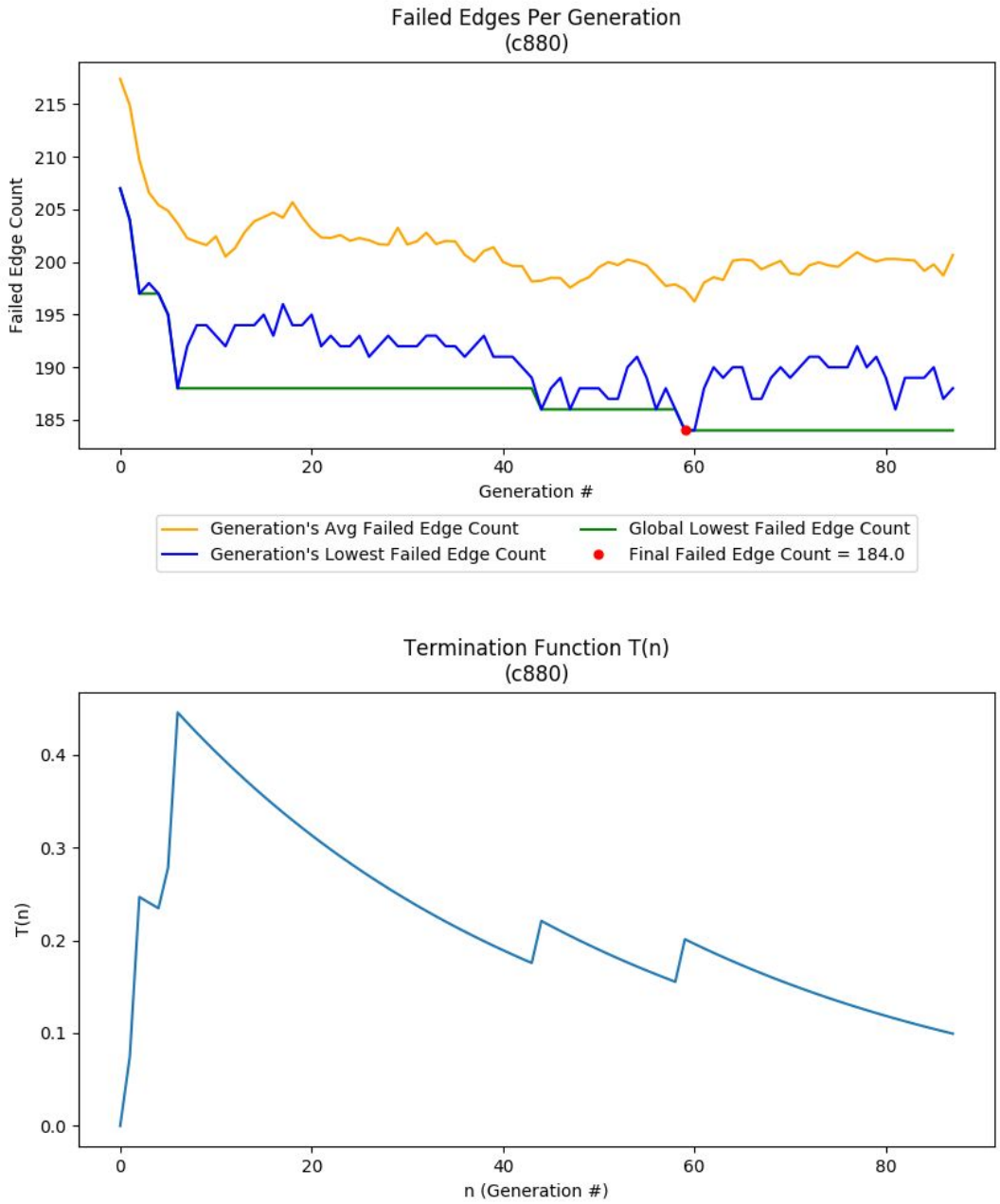


Figure 25. Genetic Results for ISCAS Test c880.

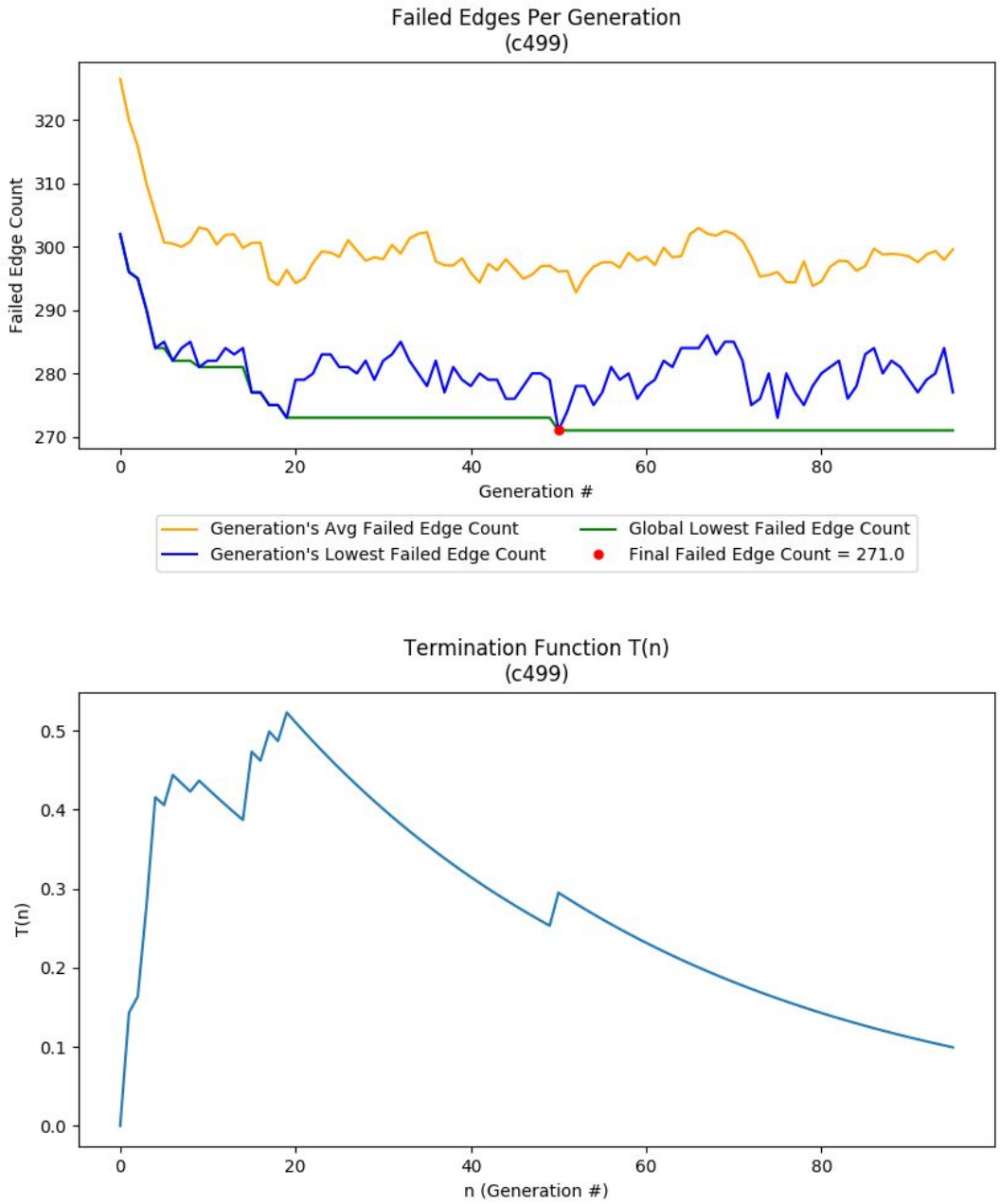


Figure 26. Genetic Results for ISCAS Test c499.

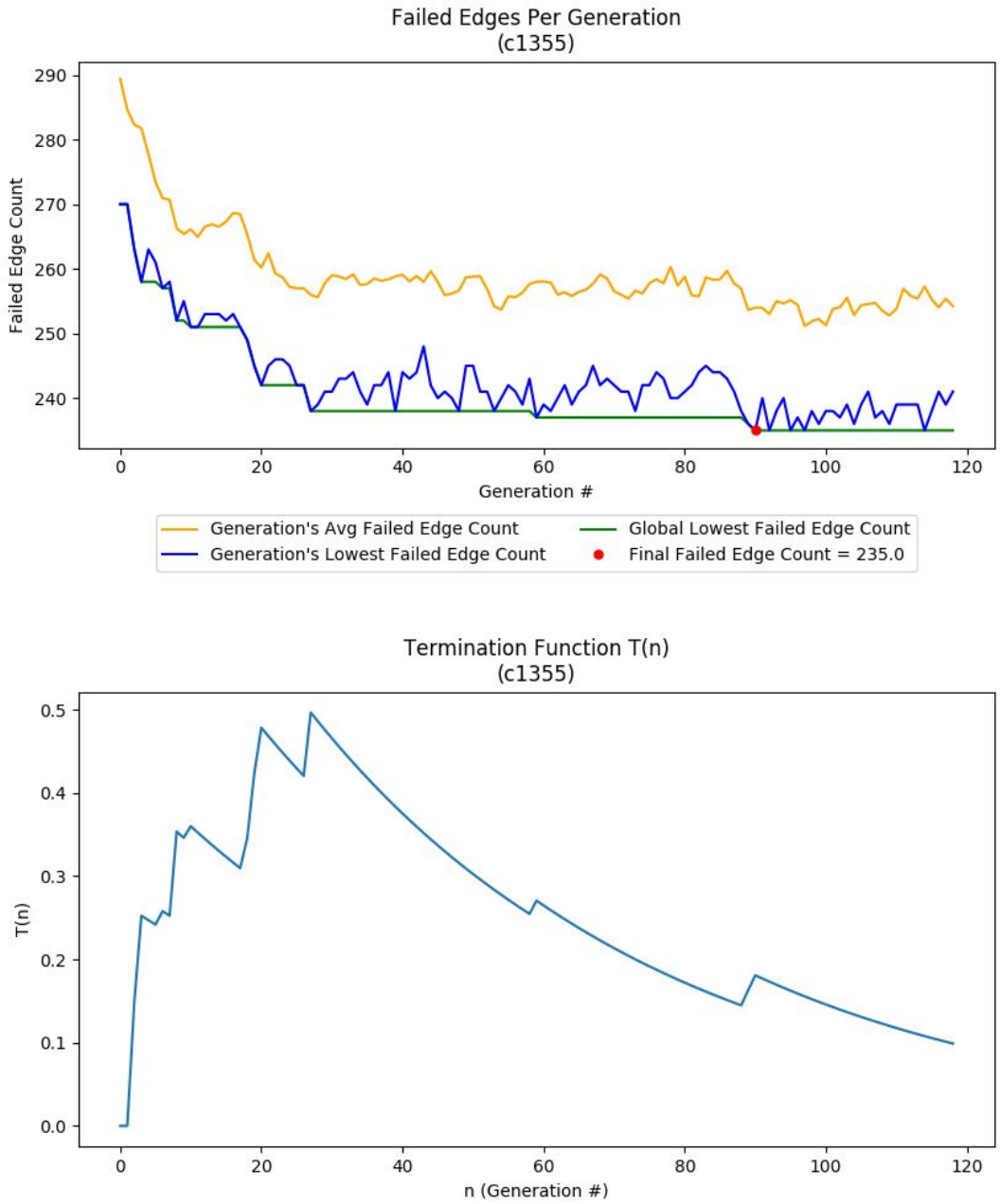
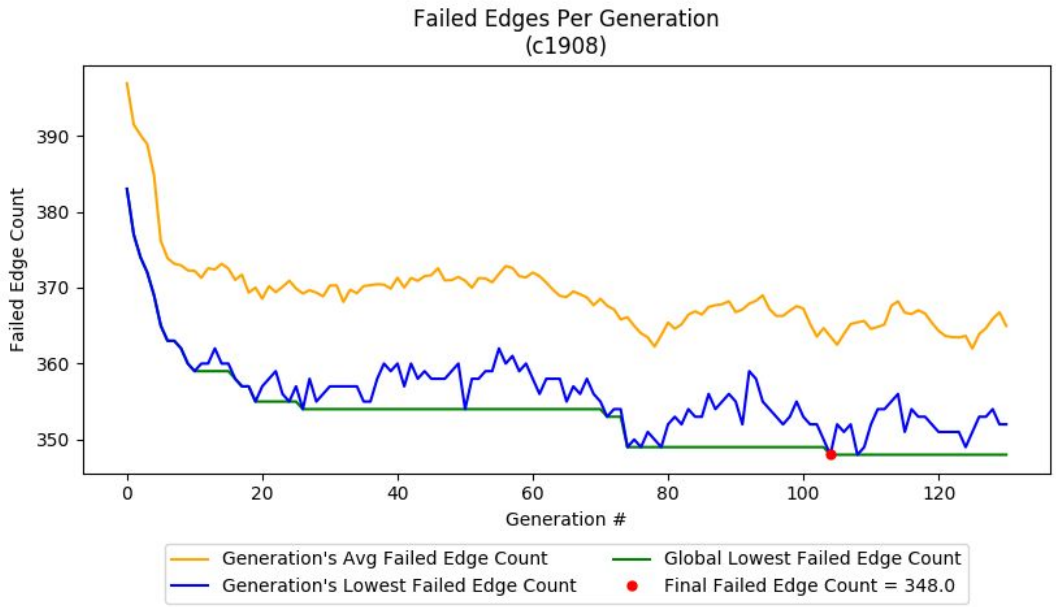


Figure 27. Genetic Results for ISCAS Test c1355.



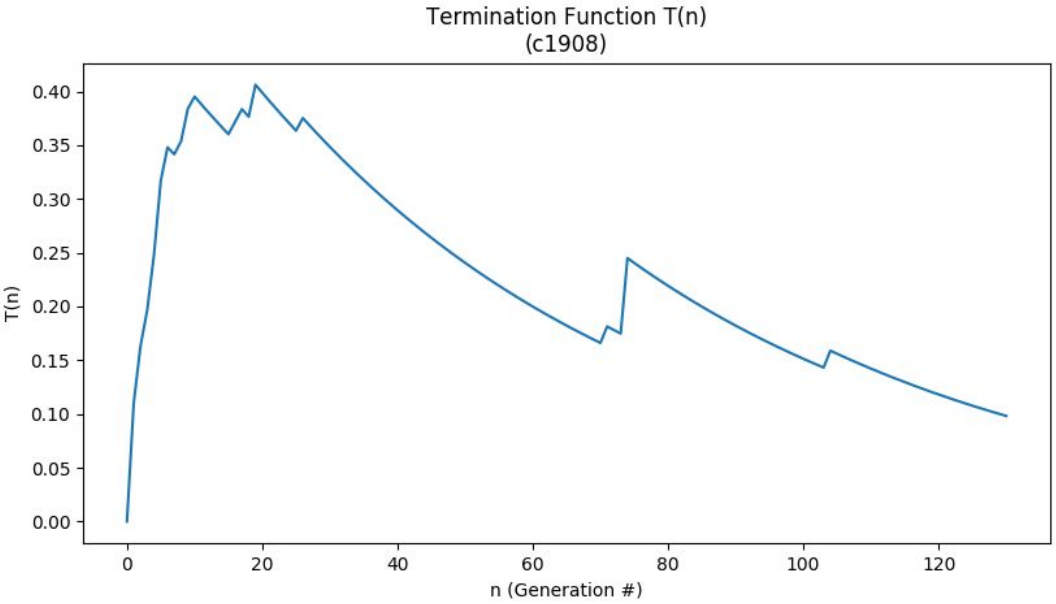
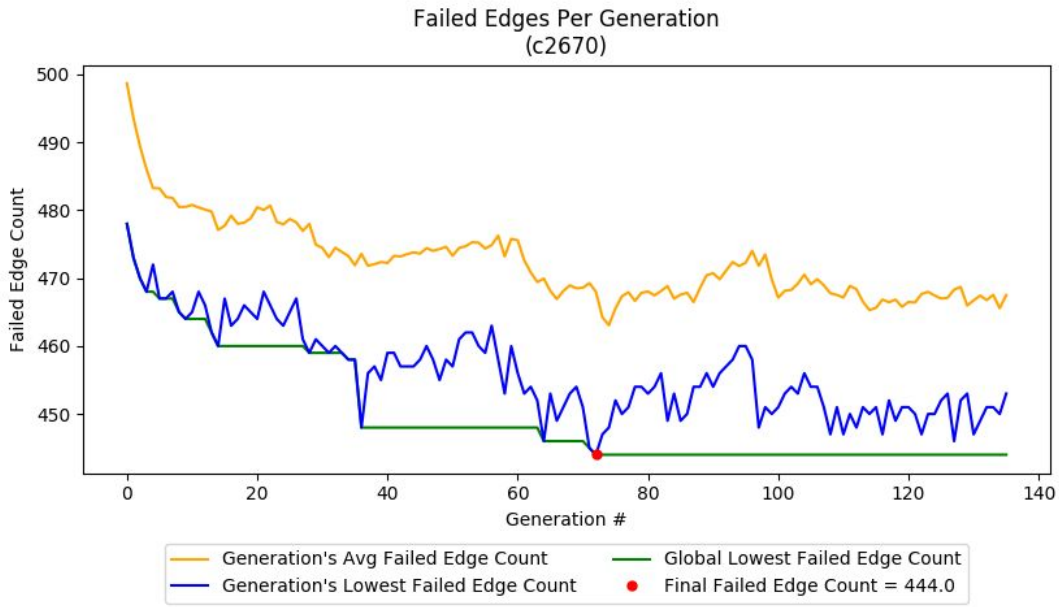


Figure 28. Genetic Results for ISCAS Test c1908.



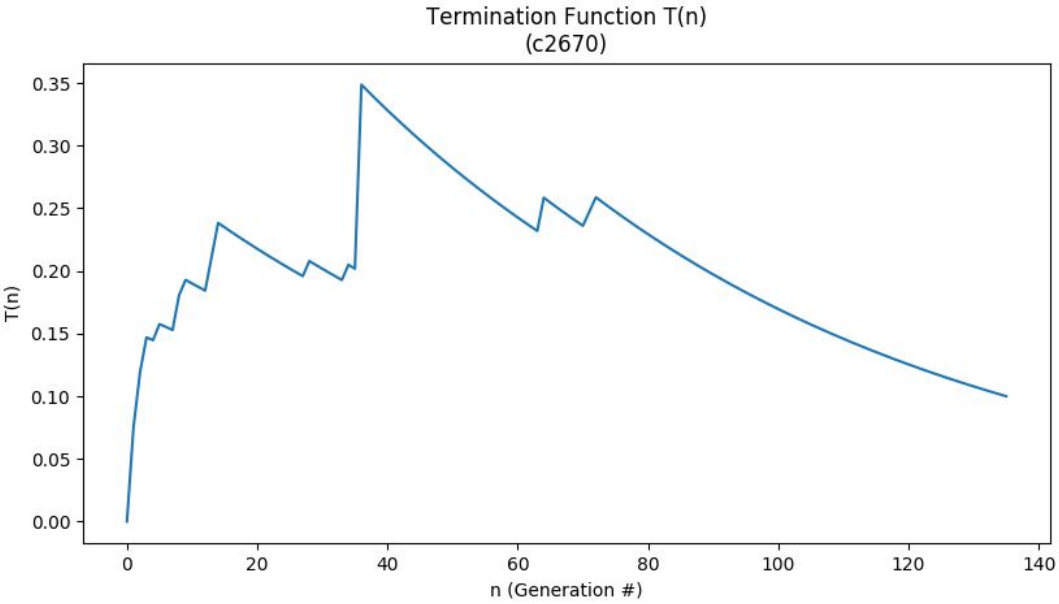


Figure 29. Genetic Results for ISCAS Test c2670.

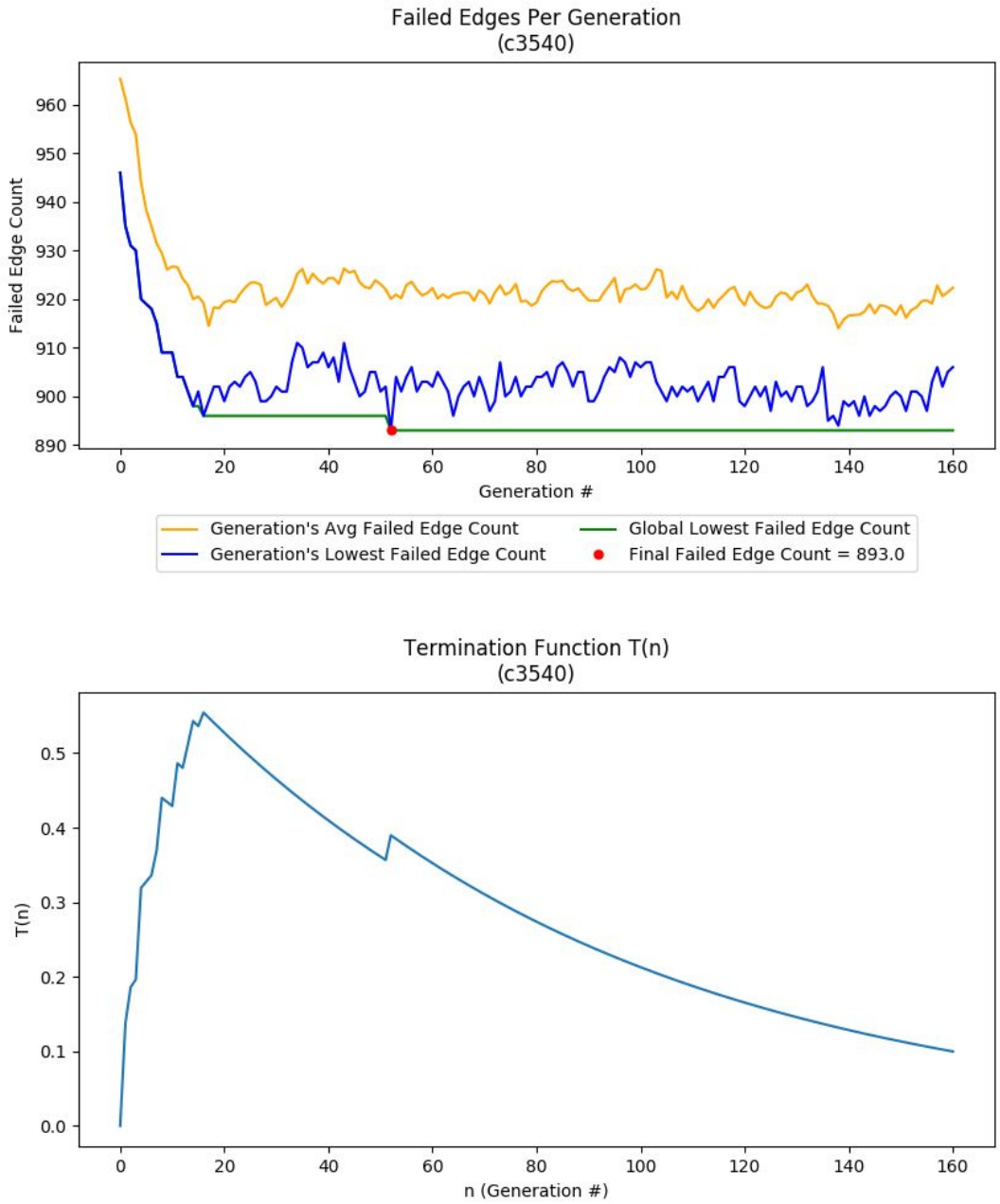
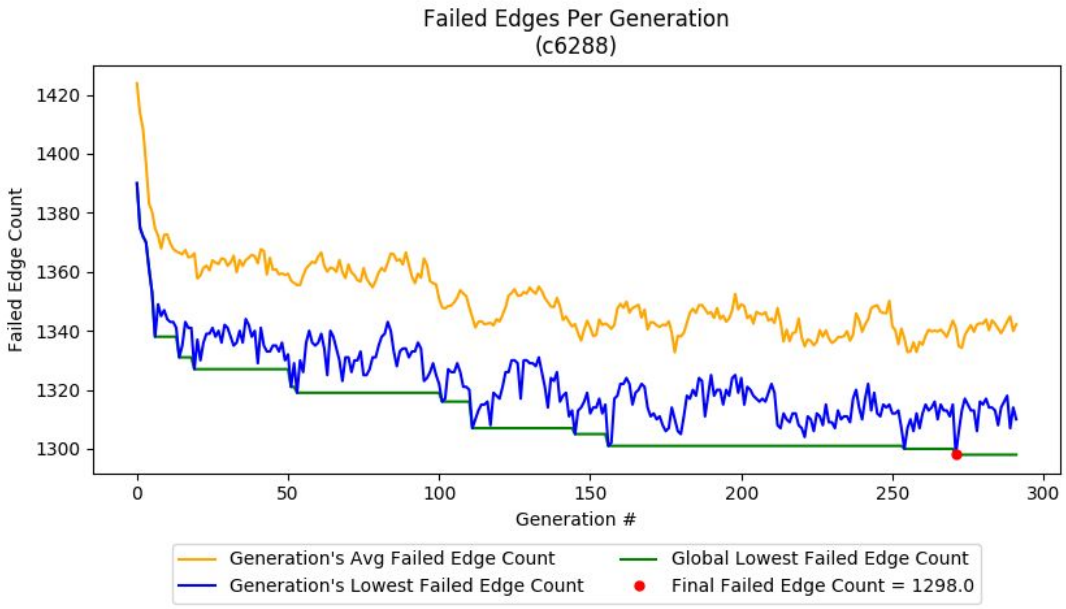


Figure 30. Genetic Results for ISCAS Test c3540.



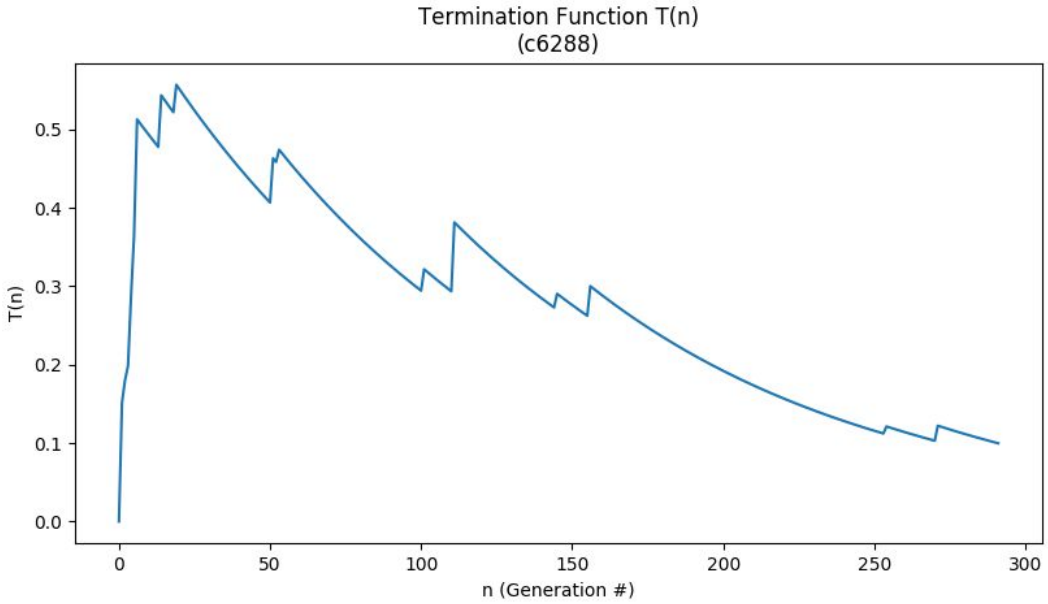


Figure 31. Genetic Results for ISCAS Test c6288.

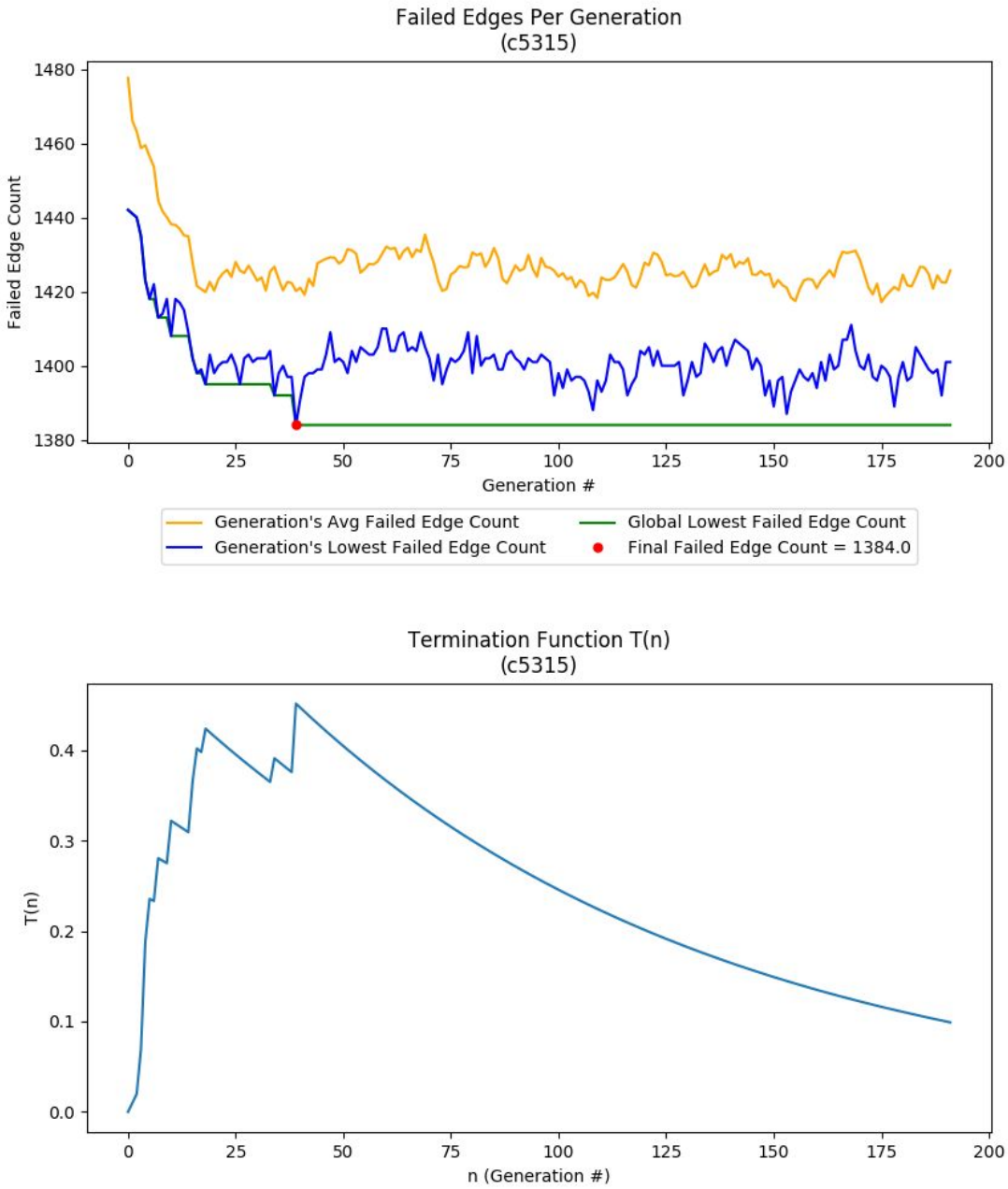
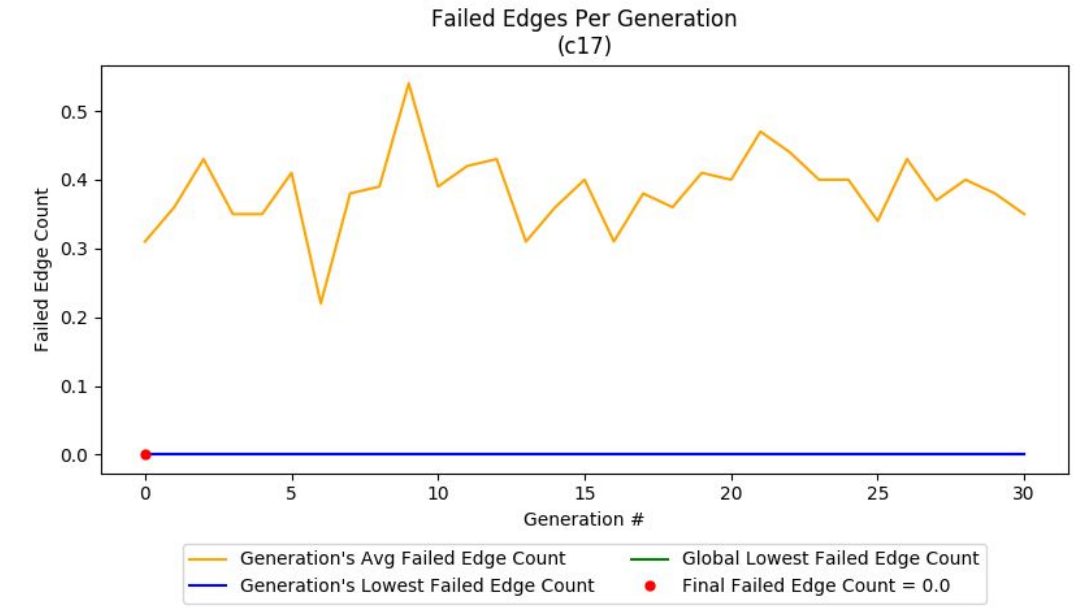


Figure 32. Genetic Results for ISCAS Test c5315.

APPENDIX B: TESTS FOR RANDOM SEARCH WITH STOP RATE CAP



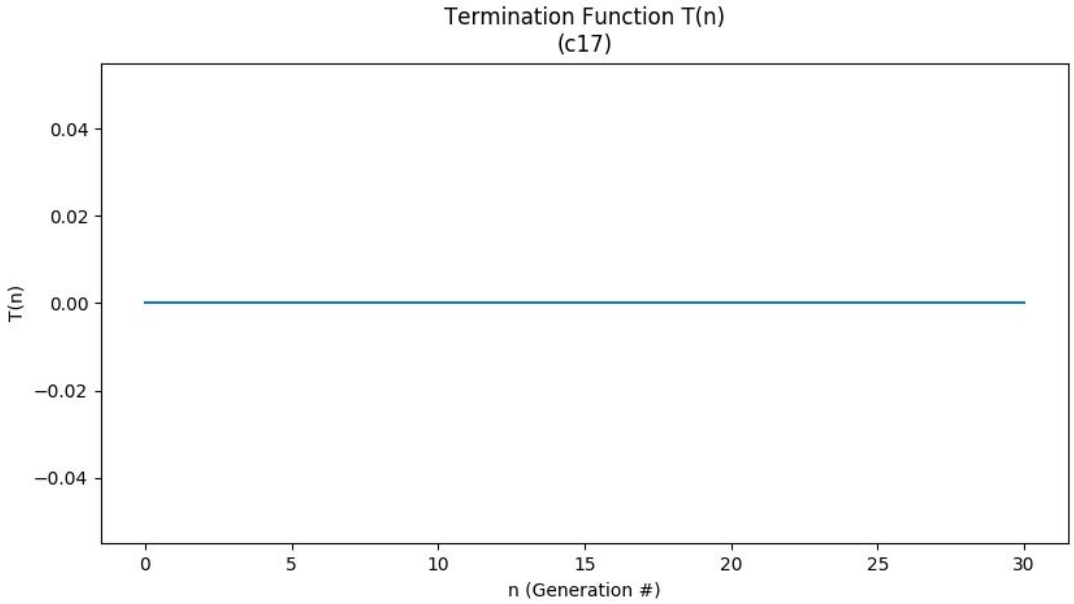
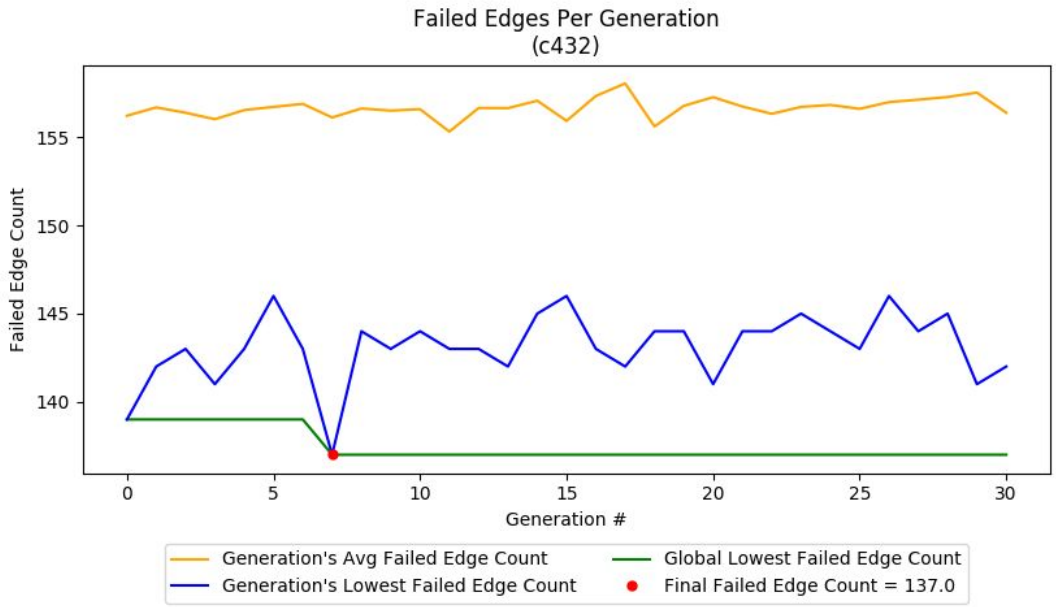


Figure 33. Random Search with Minimum Stop Rate 0.1 for ISCAS Test c17.



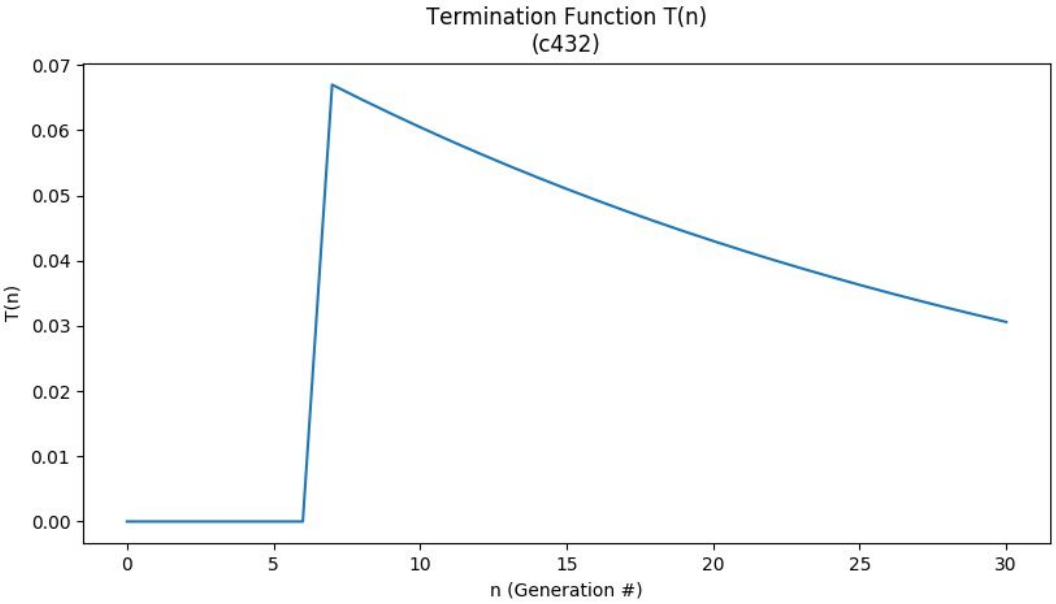


Figure 34. Random Search with Minimum Stop Rate 0.1 for ISCAS Test c432.

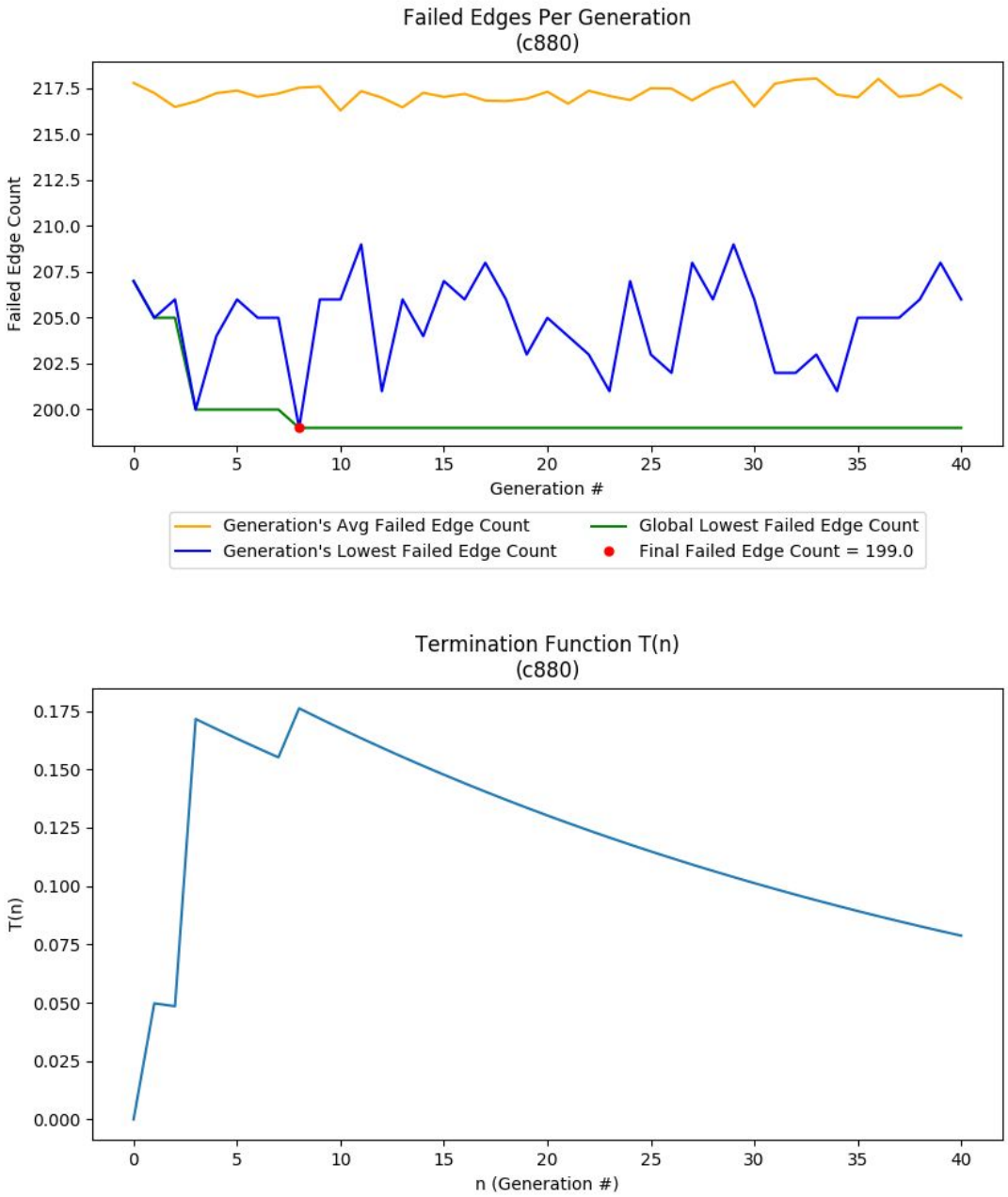


Figure 35. Random Search with Minimum Stop Rate 0.1 for ISCAS Test c880.

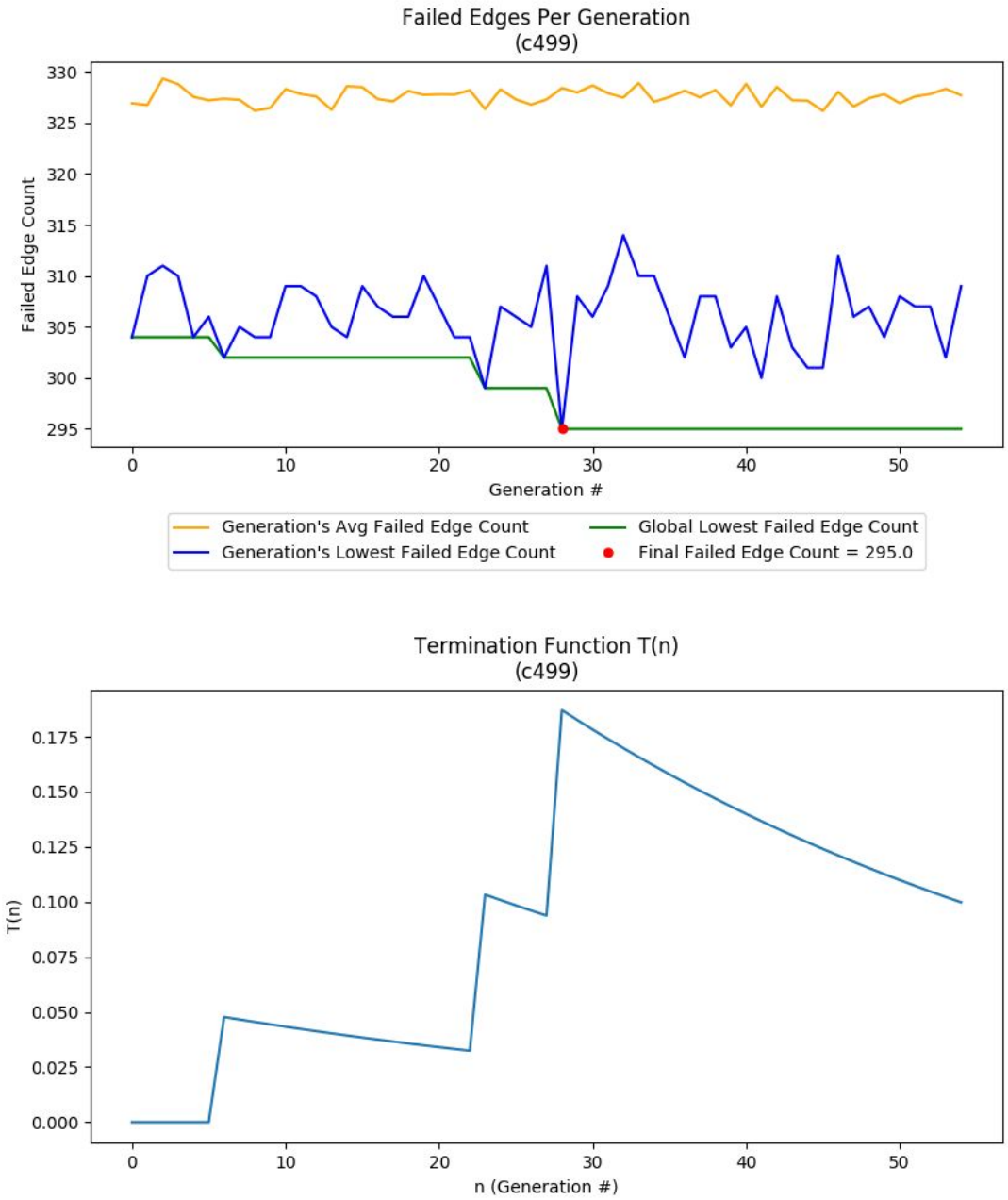


Figure 36. Random Search with Minimum Stop Rate 0.1 for ISCAS Test c499.

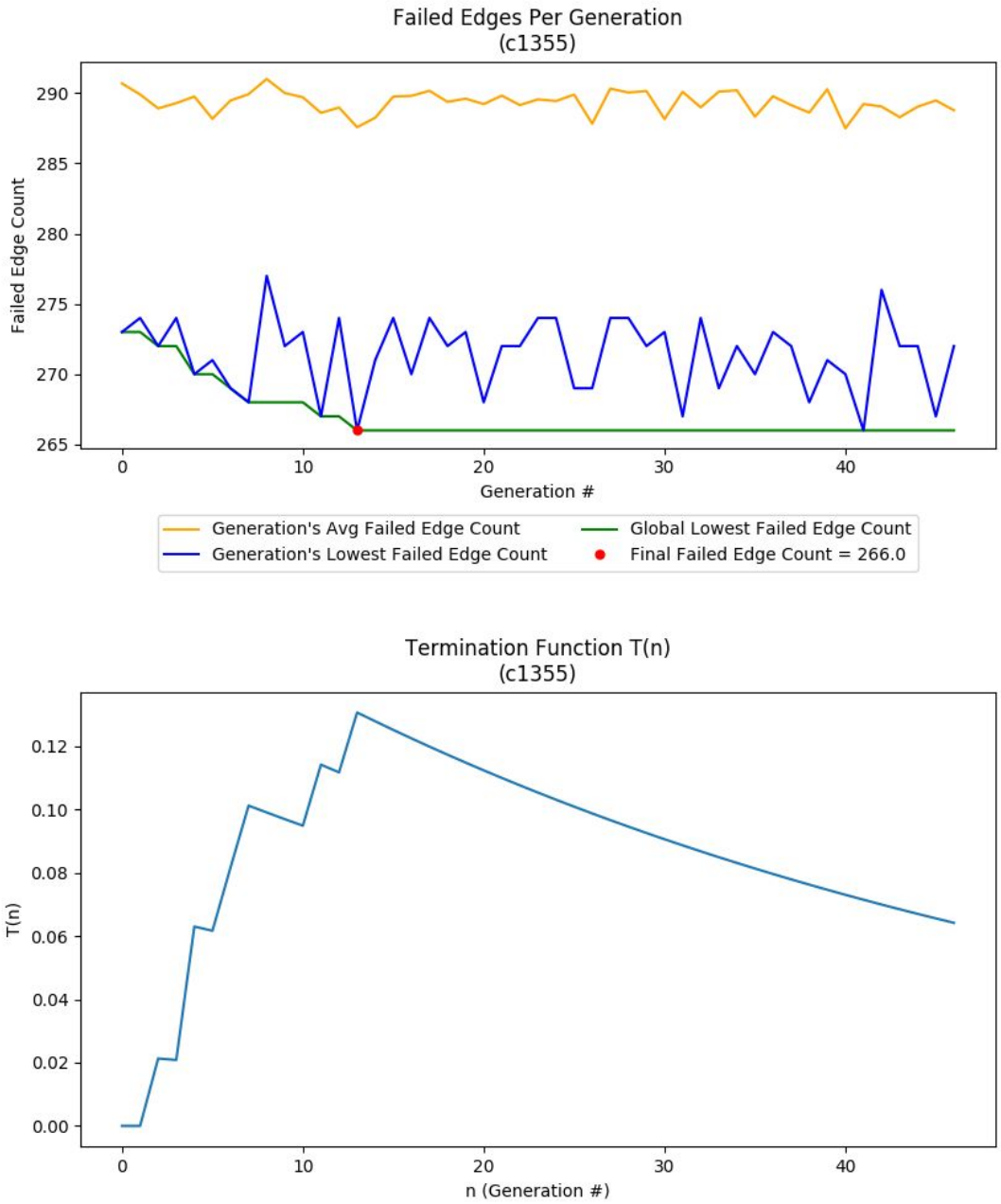


Figure 37. Random Search with Minimum Stop Rate 0.1 for ISCAS Test c1355.

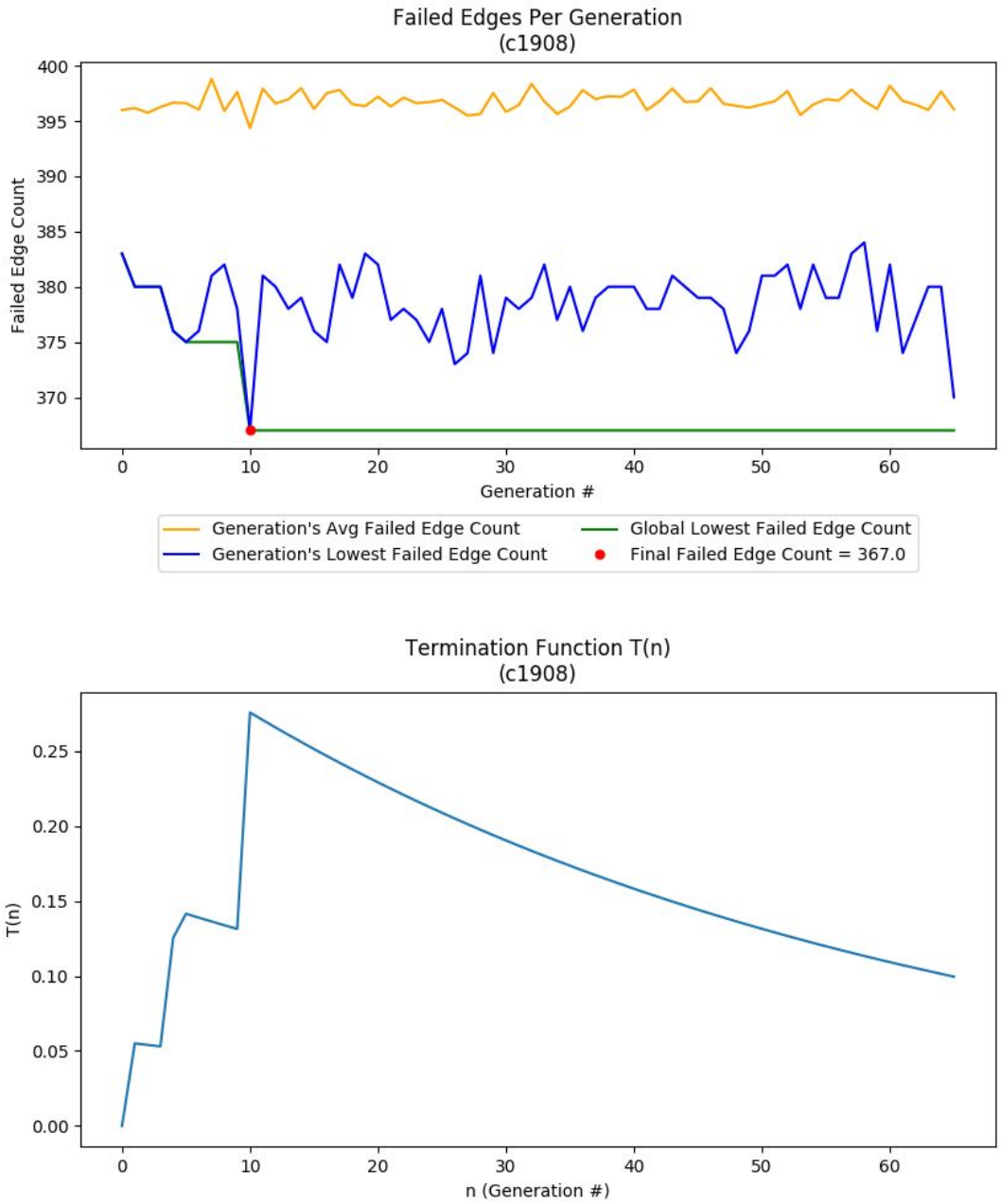


Figure 38. Random Search with Minimum Stop Rate 0.1 for ISCAS Test c1908.

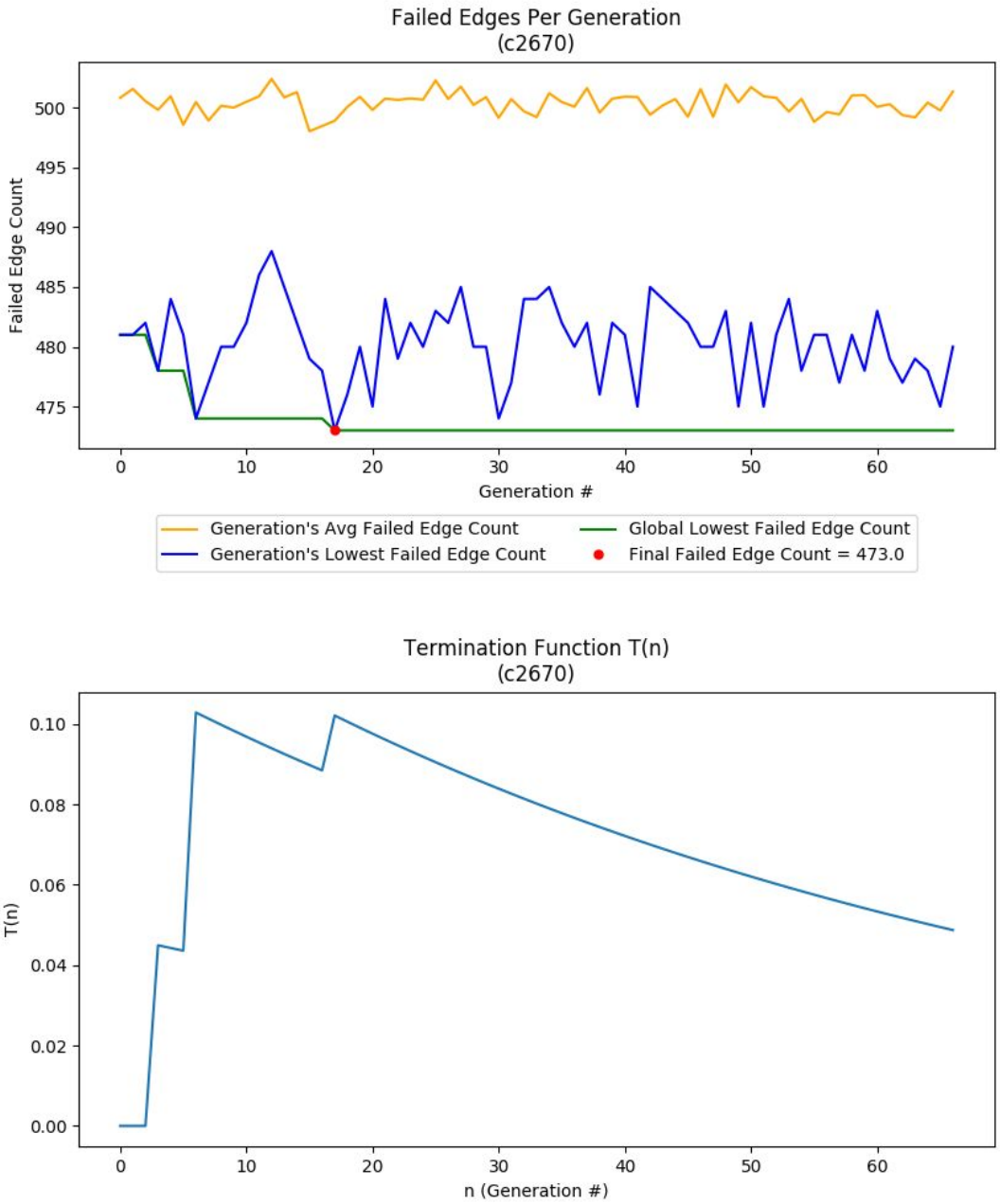
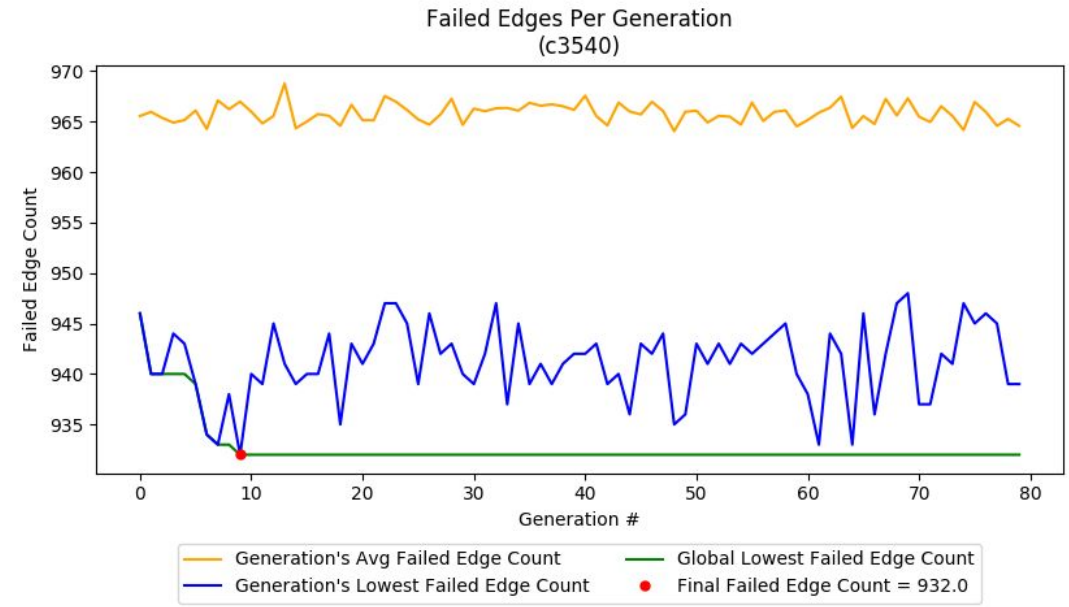


Figure 39. Random Search with Minimum Stop Rate 0.1 for ISCAS Test c2670.



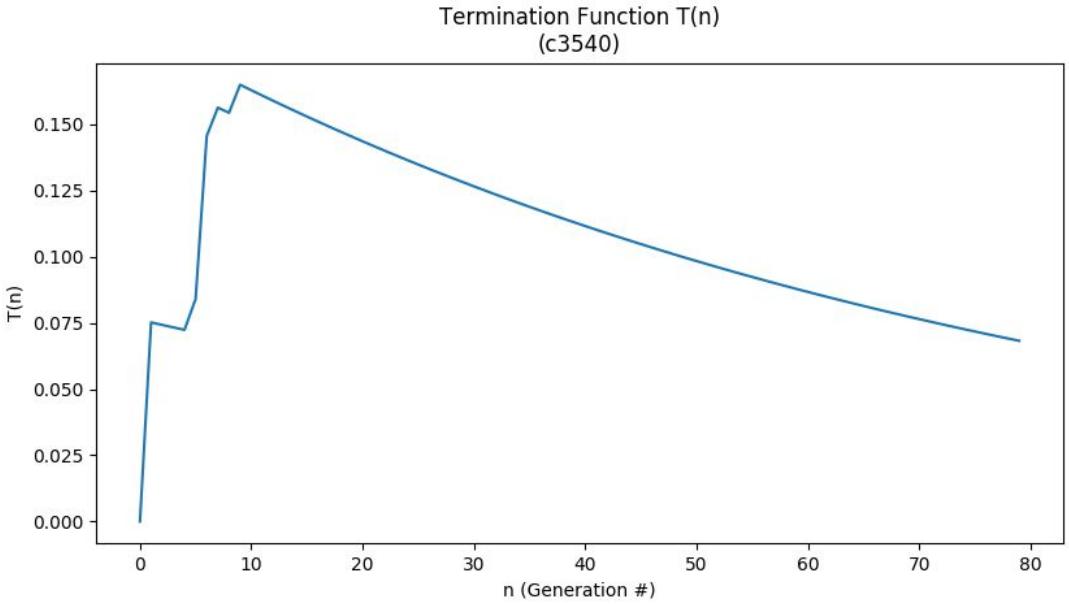


Figure 40. Random Search with Minimum Stop Rate 0.1 for ISCAS Test c3540.

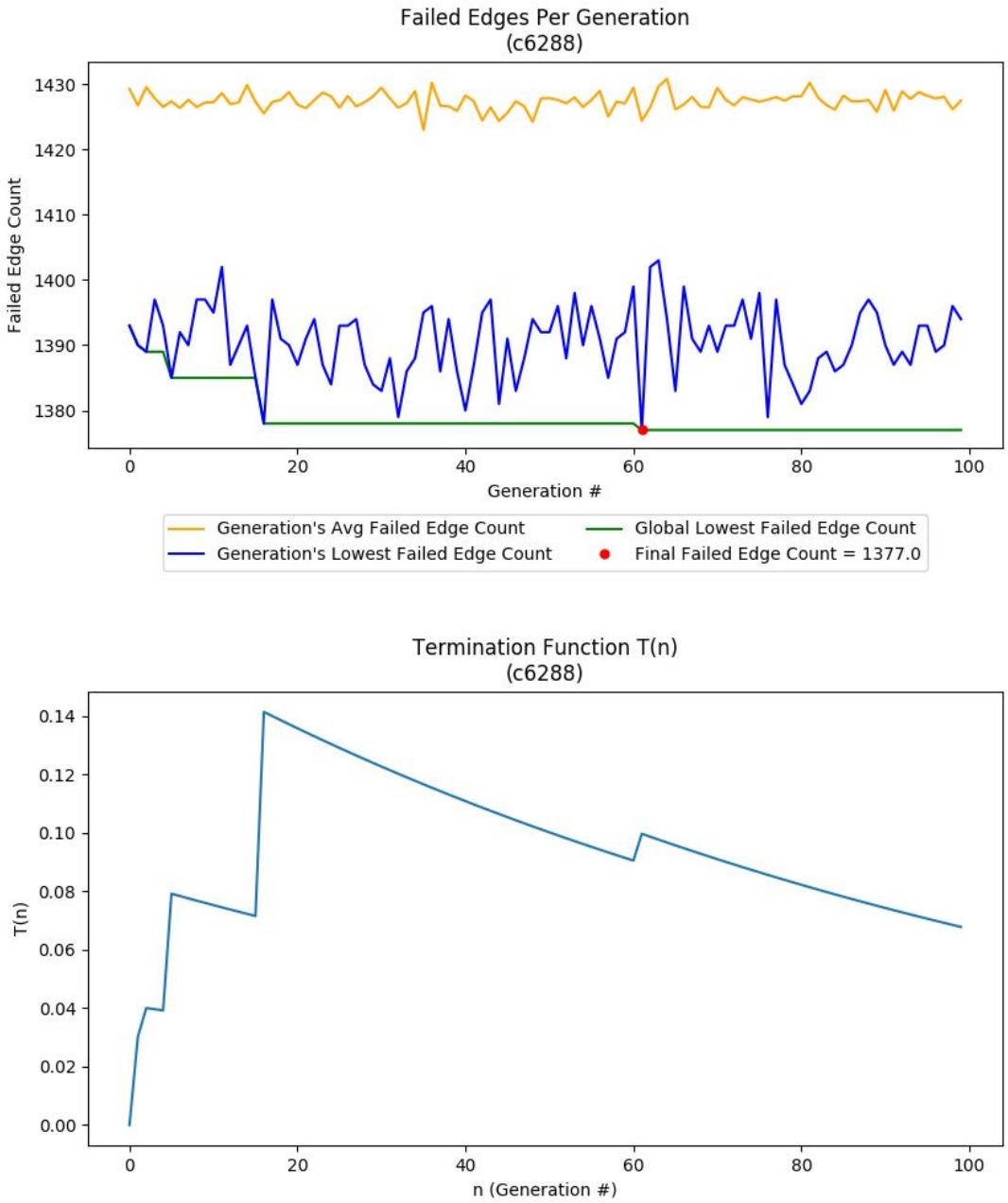


Figure 41. Random Search with Minimum Stop Rate 0.1 for ISCAS Test c6288.

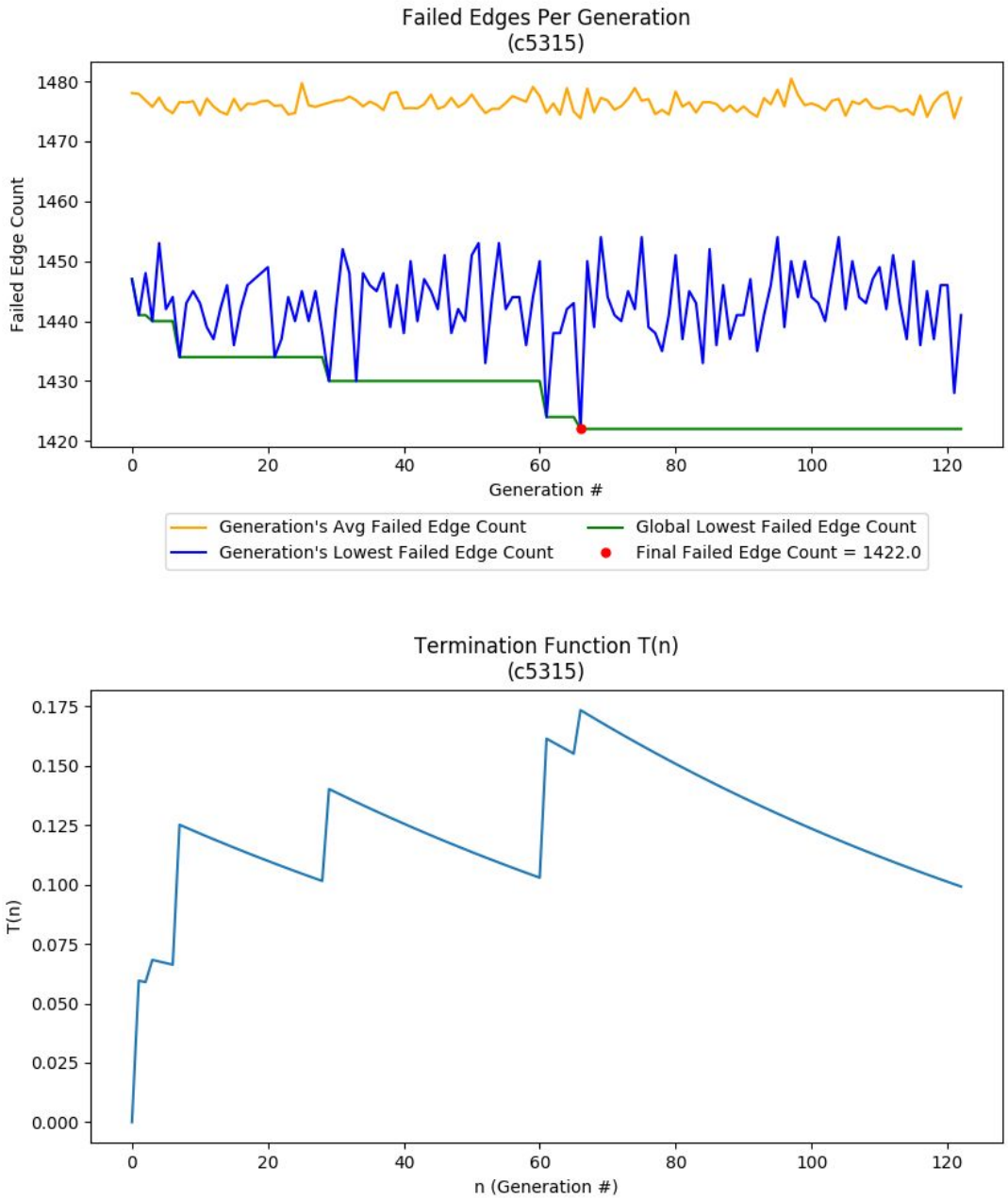


Figure 42. Random Search with Minimum Stop Rate 0.1 for ISCAS Test c5315.

APPENDIX C: TESTS FOR RANDOM SEARCH WITH ITERATION CAP

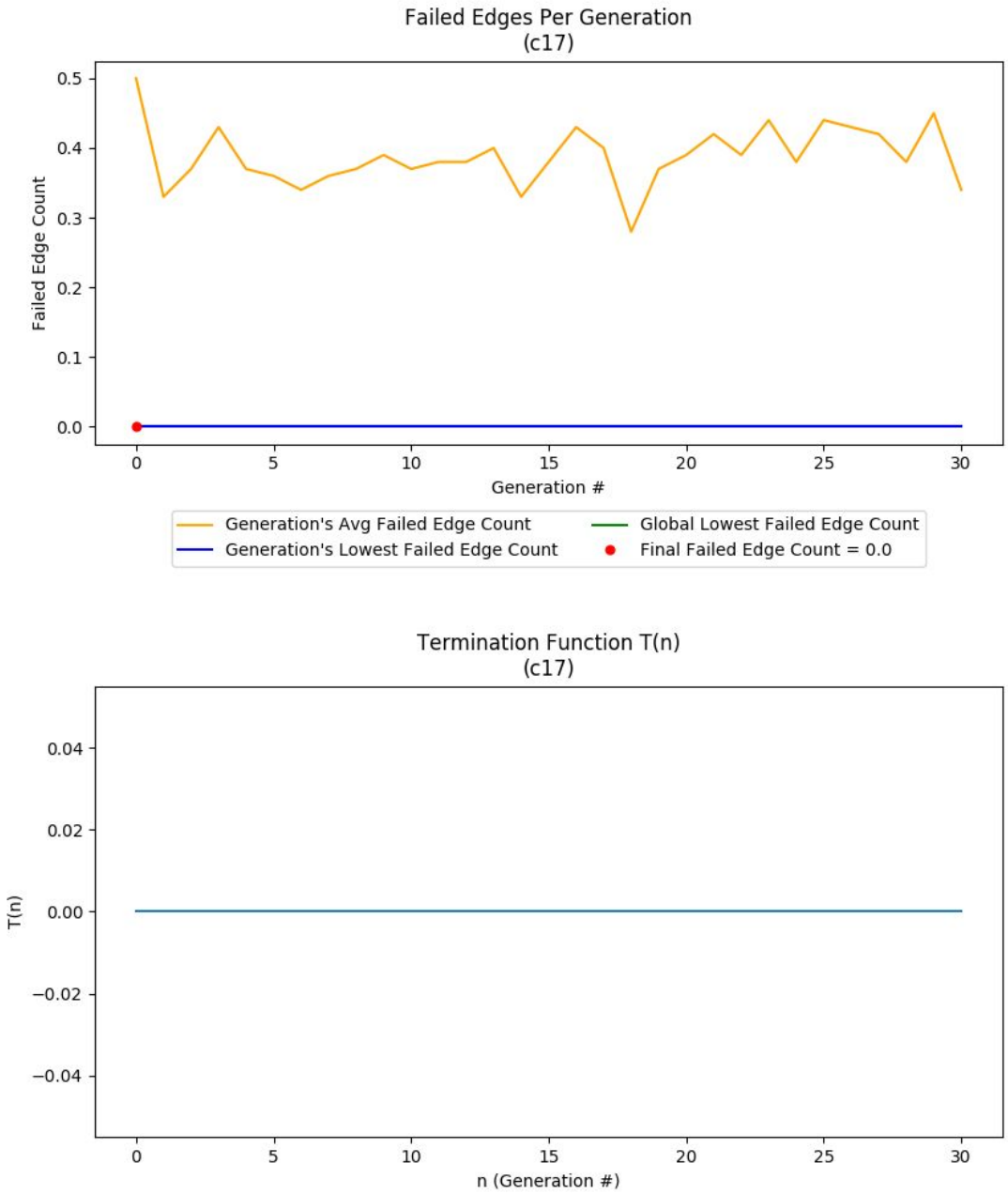


Figure 43. Random Search Results for ISCAS Test c17 with 31 Iterations.

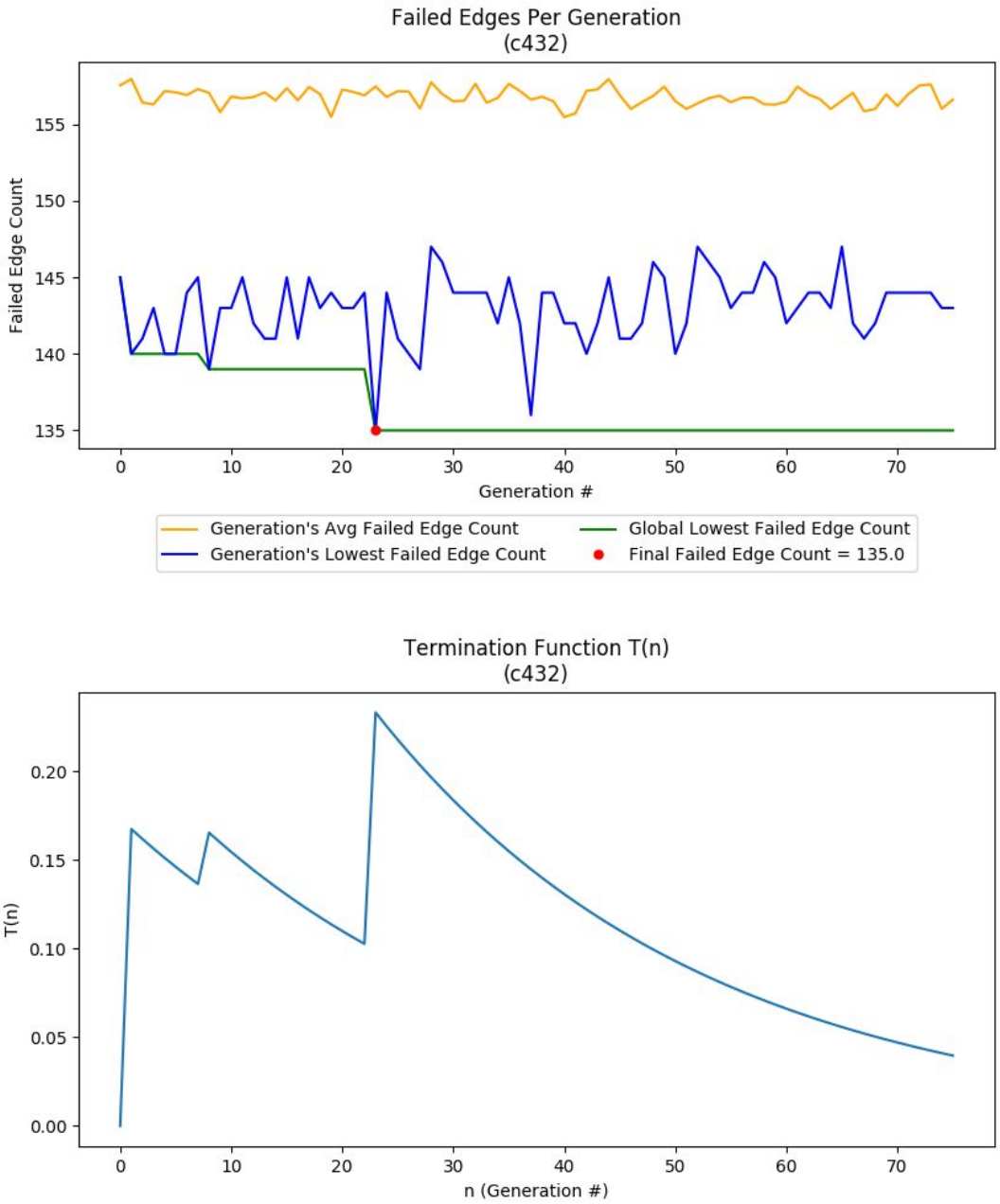
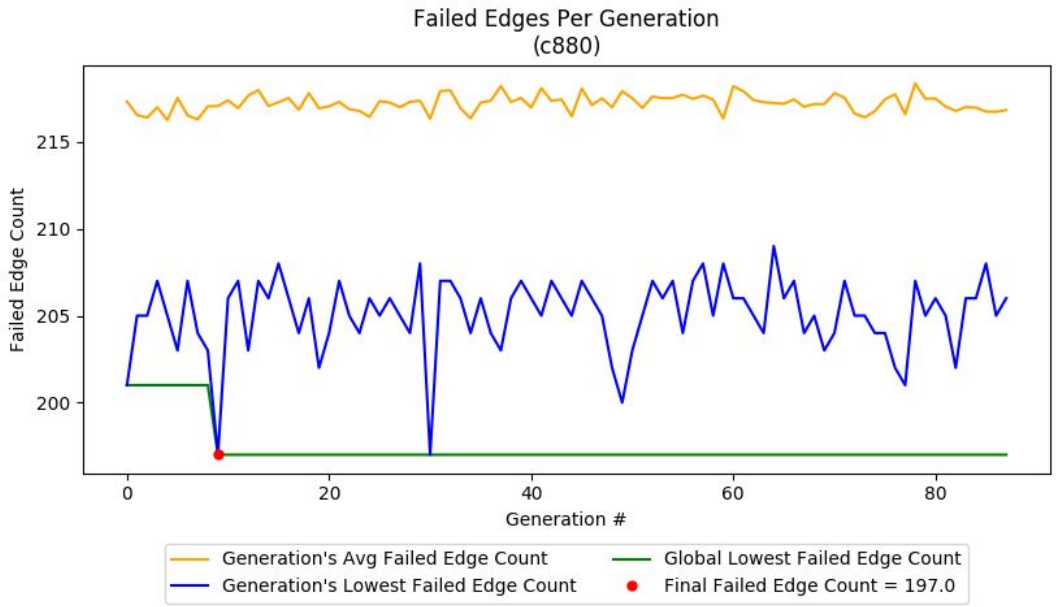


Figure 44. Random Search Results for ISCAS Test c432 with 76 Iterations.



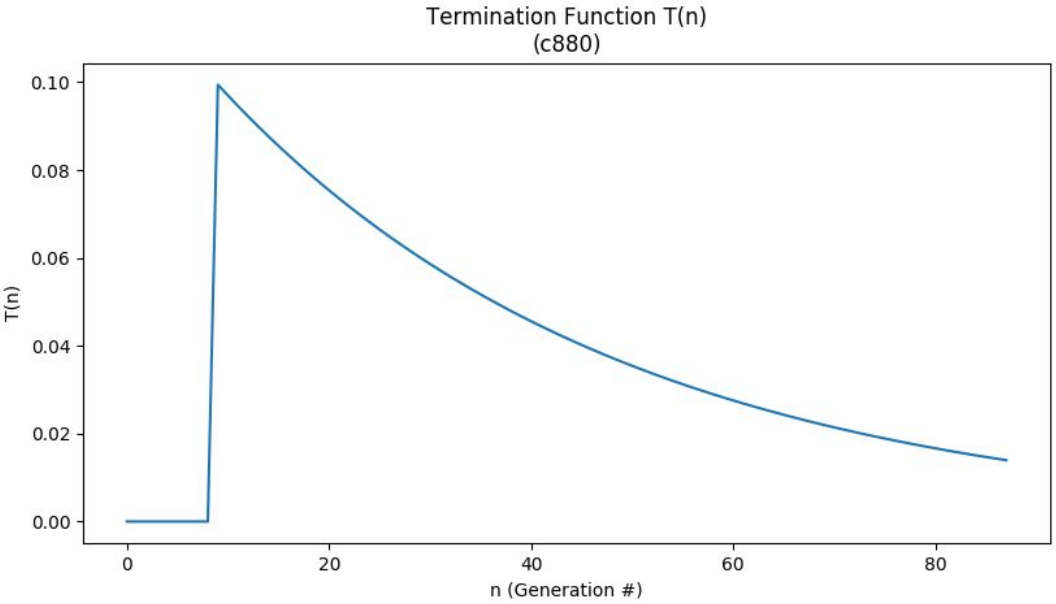


Figure 45. Random Search Results for ISCAS Test c880 with 88 Iterations.

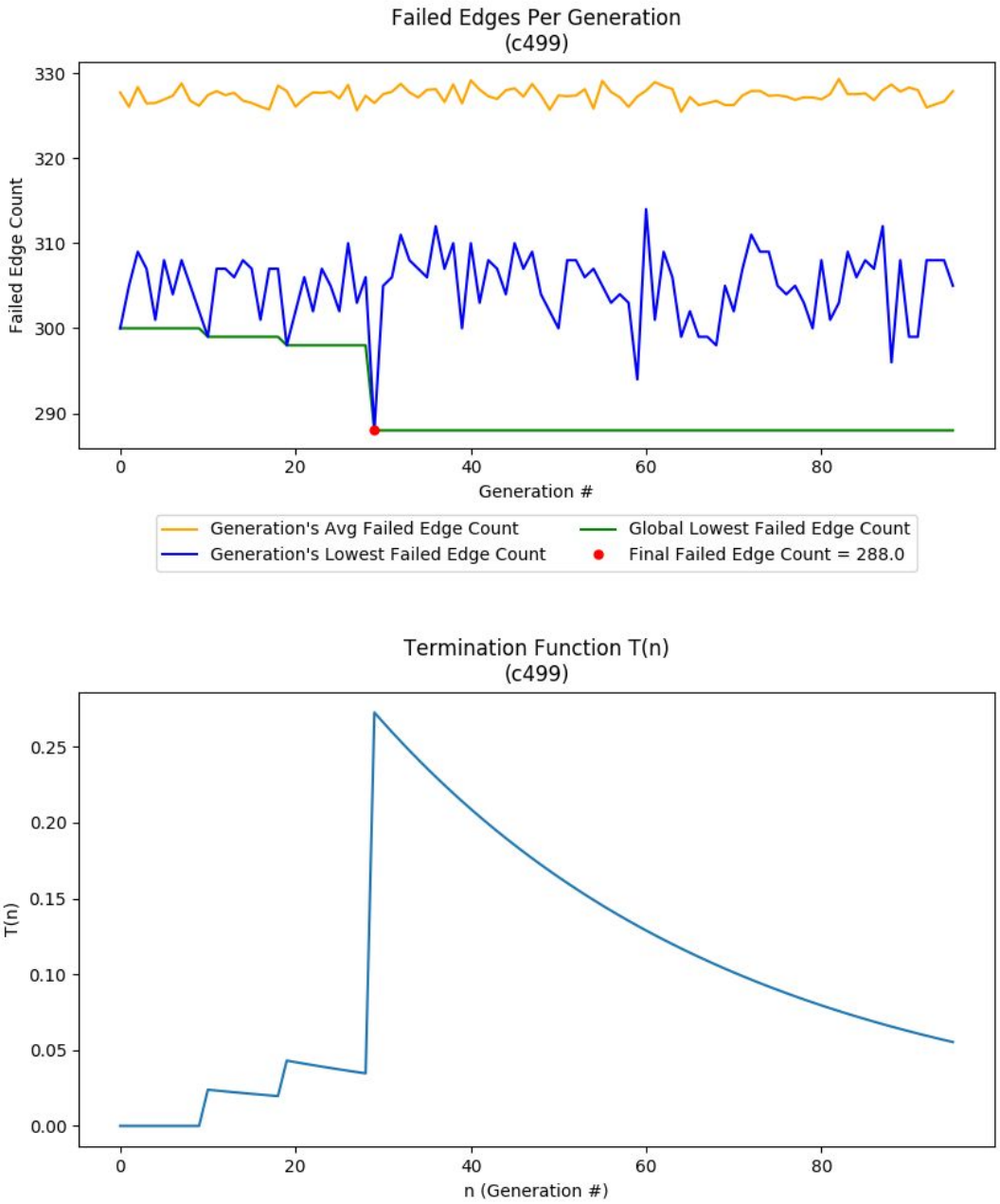


Figure 46. Random Search Results for ISCAS Test c499 with 96 Iterations.

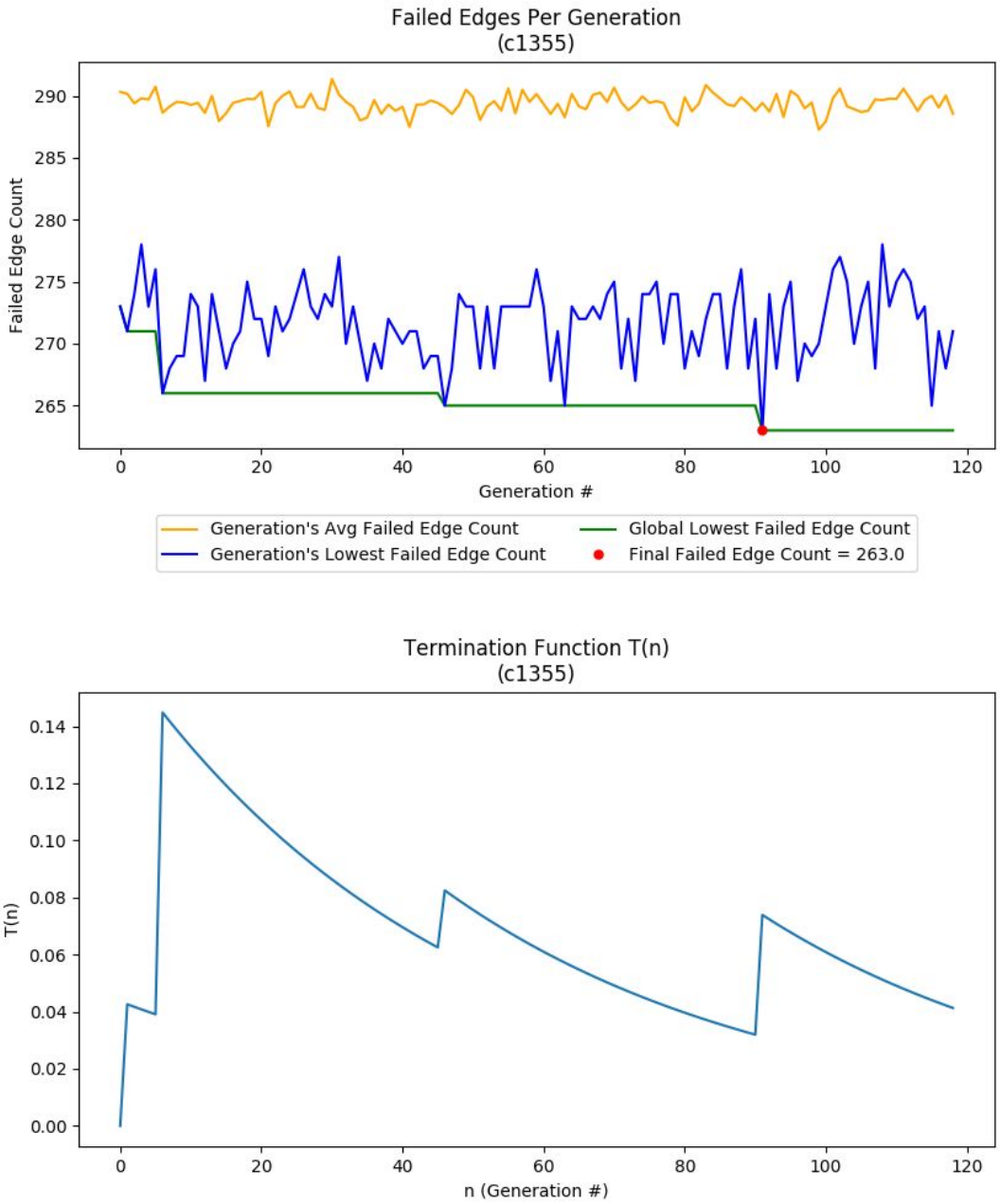


Figure 47. Random Search Results for ISCAS Test c1355 with 119 Iterations.

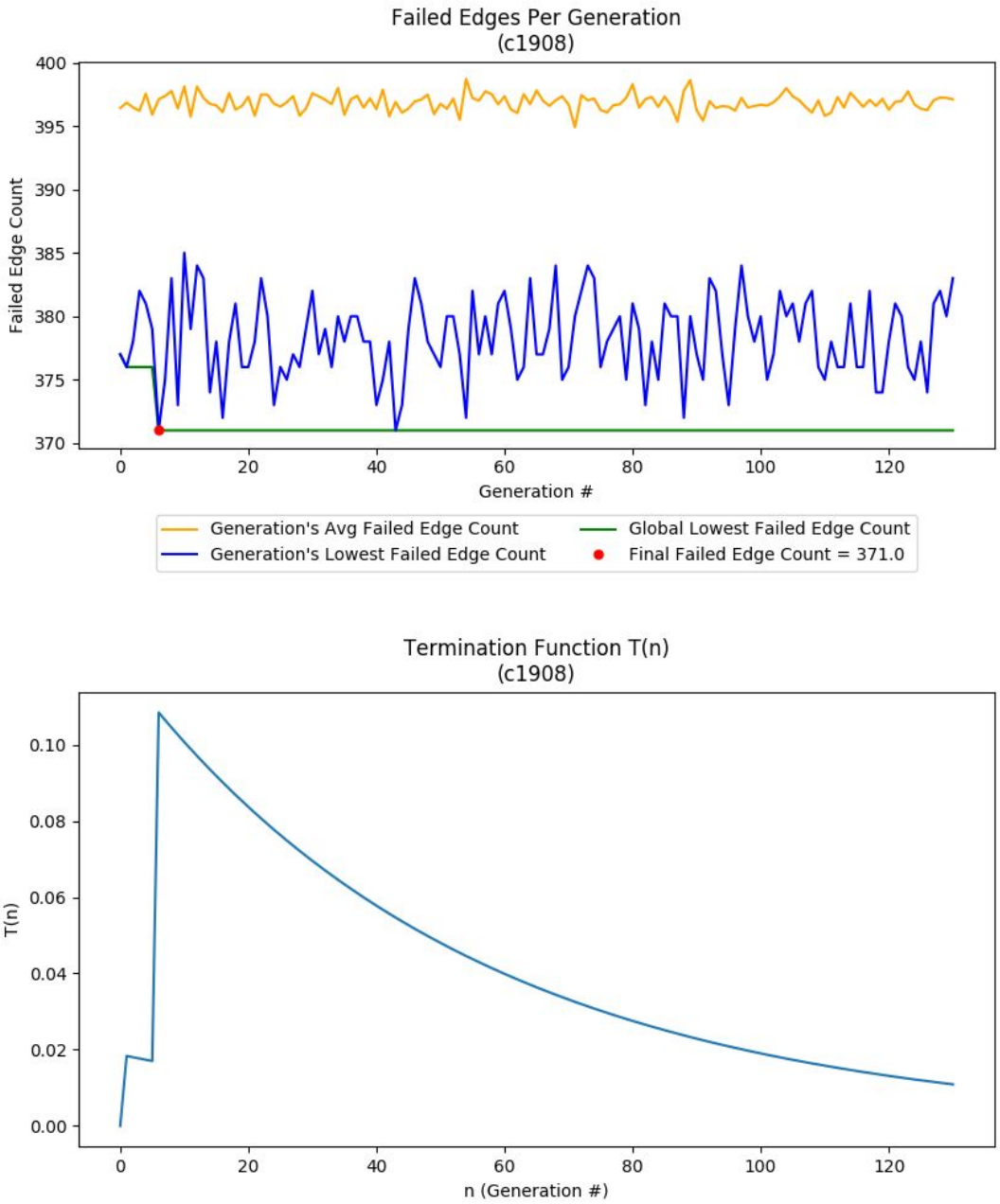


Figure 48. Random Search Results for ISCAS Test c1908 with 131 Iterations.

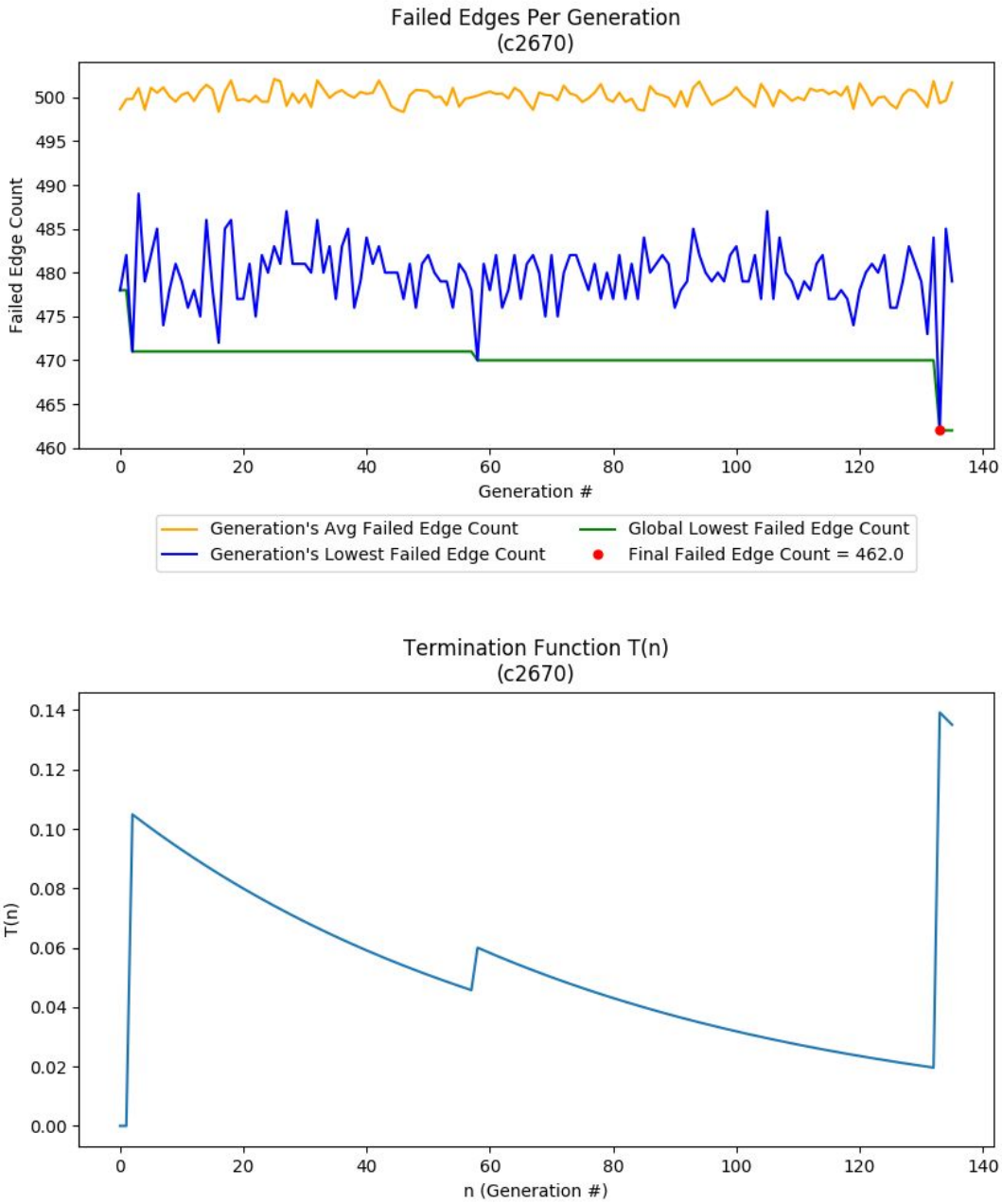


Figure 49. Random Search Results for ISCAS Test c2670 with 136 Iterations.

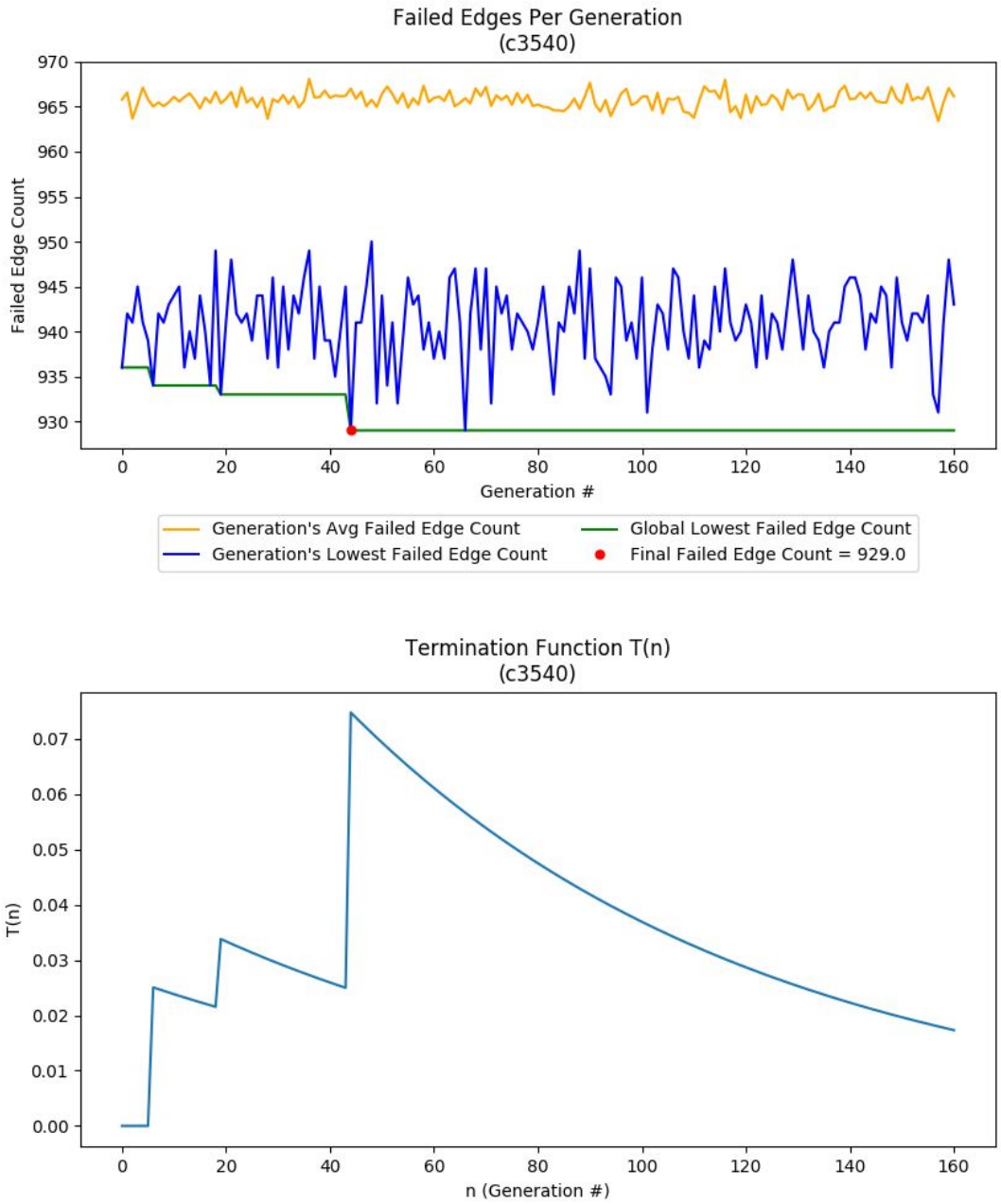


Figure 50. Random Search Results for ISCAS Test c3540 with 161 Iterations.

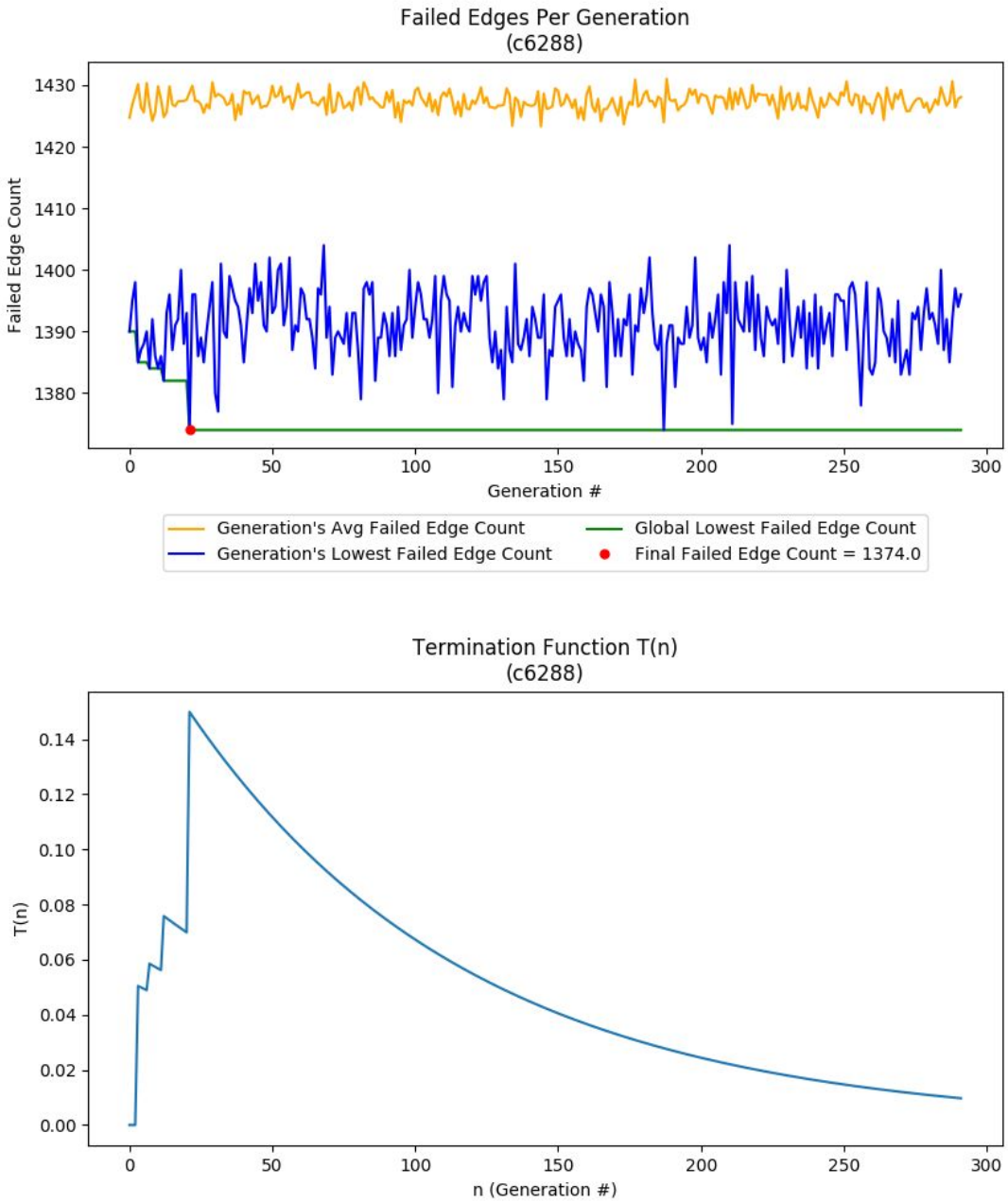


Figure 51. Random Search Results for ISCAS Test c6288 with 292 Iterations.

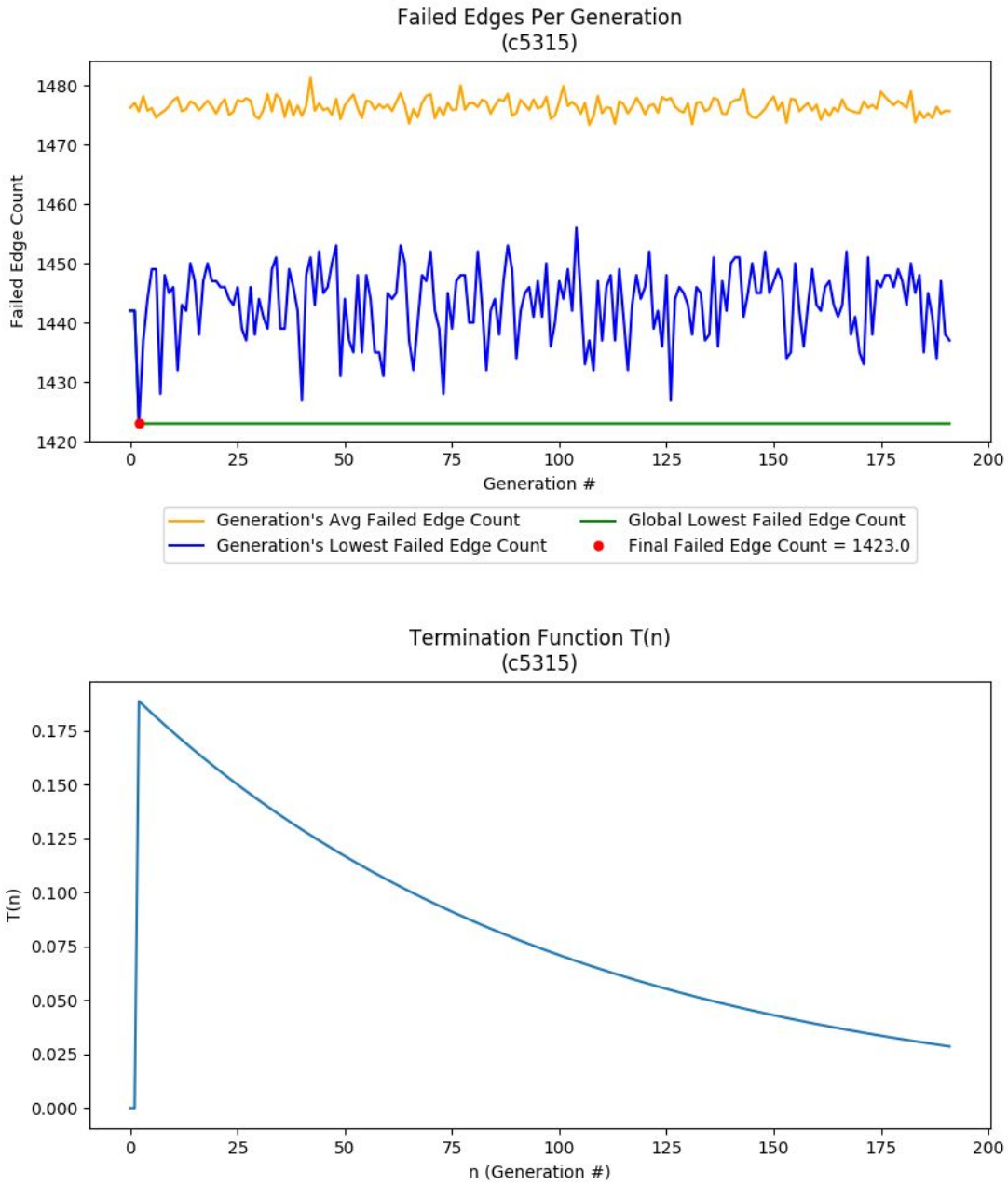


Figure 52. Random Search Results for ISCAS Test c5315 with 192 Iterations.

VITA

Alexander Gunter grew up in Jackson, Mississippi and entered Ole Miss for computer science and mathematics in 2012. He worked as a lab assistant for the Department of Computer and Information Science before becoming an assistant network administrator for the department. During this time he joined the Ole Miss VLSI System Research Design Laboratory to research behavioral modelling of nanomagnet logic devices, and he investigated marker tracking with the Ole Miss High Fidelity Virtual Environments Lab. He was awarded the 2015-2016 T.A. Bickerstaff Scholarship for achievement in the Department of Mathematics and the 2016-2017 Richard E. Grove Award for service to the Department of Computer and Information Science. He graduated magna cum laude from the University of Mississippi in December of 2016, receiving the degree of Bachelor of Science in Computer Science. He entered the Graduate School at the University of Mississippi in 2017 to continue his research with the Department of Electrical Engineering. He worked as a teaching assistant while developing design automation techniques for perpendicular nanomagnet logic.

Publications and Presentations

- *A Framework for Verification of Signal Propagation Through Sequential Nanomagnet Logic Devices*. Undergraduate thesis at the University of Mississippi.
- *Design and Analysis of Sequential Reversible Logic Structures Using Nanomagnet Logic*. Works-in-Progress poster and presentation at DAC 2015.
- *Real-Time Marker Tracking with Microsoft Kinect*. Poster and presentation at ICAT-EGVE 2016.
- *Behavioral Verification and Crossing Reduction Algorithms for pNML Devices*. Poster and presentation at DAC 2018.