Universidade Federal de Pernambuco

Centro de Informática

Aplicação de CGP na Síntese de Circuitos Tolerantes a Falhas

Humberto Costa Cordeiro Távora

Orientador: Stefan Michael Blawid

Recife, 2025

Humberto Costa Cordeiro Távora

Aplicação de CGP na Síntese de Circuitos Tolerantes a Falhas

Projeto apresentado no curso de Engenharia da Computação, como um requisito parcial para obter o Título de Bacharel em Engenharia da Computação, no Centro de Informática da Universidade Federal de Pernambuco.

Orientador: Stefan Michael Blawid

Recife, 2025

Ficha de identificação da obra elaborada pelo autor,
através do programa de geração automática do SIB/UFPE

Távora, Humberto Costa Coderio.
Aplicação de CGP na Síntese de Circuitos Tolerantes a Falhas / Humberto Costa Coderio Távora. - Recife, 2025.
36 p.

Orientador(a): Stefan Michael Blawid
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Informática, Engenharia da Computação - Bacharelado, 2025.

1. Algoritmos Evolutivos. 2. Circuitos Lógicos Combinacionais. 3. Programação Genética Cartesiana. I. Blawid, Stefan Michael. (Orientação). II. Título.

000 CDD (22.ed.)

Humberto Costa Cordeiro Távora

APLICAÇÃO DE CGP NA SÍNTESE DE CIRCUITOS TOLERANTES A FALHAS

Trabalho de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia da Computação da Universidade Federal de Pernambuco, como requisito parcial para obtenção do título de bacharel em Engenharia da Computação.

Aprovado em: 03/04/2025

BANCA EXAMINADORA

_____

Prof. Dr. Stefan Michael Blawid (Orientador)

Universidade Federal de Pernambuco

_____

Prof. Dr. Paulo Salgado Gomes de Mattos Neto (Examinador Interno)

Universidade Federal de Pernambuco

Agradecimentos

Eu gostaria de agradecer a todos que me ajudaram durante esta jornada, especialmente:

Aos meus pais, Patricia Costa e Henrique Távora, por todo amor, cuidado e orientação desde que vim ao mundo. Sem o sacrifício e dedicação deles, este trabalho não seria possível. Aos meus tios e tias Paloma Costa e Fred Dominguez, por todo a apoio, carinho e suporte aos meus estudos. Talvez sem o primeiro livro de Harry Potter que ganhei de minha tia na infância, meu gosto pela leitura e caminho acadêmico não fosse o mesmo. Á minha tia Maria do Carmo, pelo carinho, suporte, e esforço em cada etapa do meu crescimento, graças a ele estou finalizando mais etapa.

Às minhas avós Maria José e Sônia dos Reis, por terem me criado, educado, orientado, e todo amor e carinho que somente as avós podem dar.

Aos meus amigos Benildes, Coxinha, Fialho, João, Pedro Henrique e Tiago, pelos momentos especiais que compartilhamos nessa jornada, pelo apoio emocional e risadas que tornaram o caminho até aqui mais fácil.

À minha esposa, Bruna Cabral. O seu carinho e suporte foram fundamentais para a conclusão desta etapa.

Por fim, ao meu orientador, Prof. Stefan Blawid. Um professor e pesquisador brilhante, o qual tenho a honra de ser orientado. Sem sua orientação, paciência e conselhos, este trabalho não seria possível.

RESUMO

O avanço tecnológico tem permitido a miniaturização de circuitos, aumentando a densidade de transistores e portas lógicas. No entanto, essa complexidade crescente também eleva a suscetibilidade a defeitos, exigindo novas abordagens para o design de circuitos digitais robustos. A Programação Genética Cartesiana (CGP) surge como uma técnica promissora, capaz de evoluir circuitos lógicos com desempenho superior, especialmente em cenários onde a tolerância a falhas é crítica. Este trabalho aplicou a CGP na evolução de circuitos combinacionais básicos, como geradores de paridade ímpar, simulando falhas de componentes durante o processo evolutivo. A estratégia $(1+\lambda)$ foi utilizada para guiar a evolução, com a injeção de falhas no cálculo do *fitness* e a introdução de ruído estocástico para evitar a estagnação em máximos locais. Os resultados demonstraram que os circuitos evoluídos apresentaram maior robustez em comparação com soluções mínimas do estado da arte, com um aumento de 18,9% no *fitness* em cenários de falha única. Em cenários estocásticos, onde a probabilidade de falha variou de 1% a 5%, os circuitos evoluídos também se mostraram superiores, embora com diferenças menos expressivas. Apesar do aumento no tamanho dos circuitos, a abordagem mostrou-se eficaz para melhorar a tolerância a falhas, explorando conceitos como redundância e degeneração, inspirados em sistemas biológicos.

Palavras-chave: Programação Genética Cartesiana (CGP), Circuitos Tolerantes a Falhas, Redundância, Degeneração.

ABSTRACT

Technological advancements have enabled the miniaturization of circuits, increasing the density of transistors and logic gates. However, this growing complexity also raises susceptibility to defects, demanding new approaches for designing robust digital circuits. Cartesian Genetic Programming (CGP) emerges as a promising technique, capable of evolving logic circuits with superior performance, especially in scenarios where fault tolerance is critical. This work applied CGP to evolve basic combinational circuits, such as odd parity generators, simulating component failures during the evolutionary process. The $(1+\lambda)$ strategy was used to guide evolution, with fault injection in fitness calculation and the introduction of stochastic noise to avoid stagnation in local optima.

The results demonstrated that the evolved circuits exhibited greater robustness compared to state-of-the-art minimal solutions, with an 18.9% increase in fitness in single-failure scenarios. In stochastic scenarios, where the probability of failure ranged from 1% to 5%, the evolved circuits also performed better, albeit with less significant differences. Despite the increase in circuit size, the approach proved effective in improving fault tolerance, exploring concepts such as redundancy and degeneracy, inspired by biological systems.

Keywords: Cartesian Genetic Programming, Cartesian Genetic Programming, Redundancy, Degeneracy.

Lista de Figuras

Lista de Tabelas

LISTA DE SIGLAS

CGP	Programação Genética Cartesiana (Cartesian Genetic Programming)
EDA	Automação de Projeto Eletrônico (Electronic Design Automation)
NMR	Redundância Modular N (N-Modular Redundancy)
TMR	Redundância Modular Tripla (Triple Modular Redundancy)
SAF:	Falha Presa (Stuck-At Fault)
VLSI	Integração em Escala Muito Grande (Very Large Scale Integration)
DAG	Grafo Dirigido Acíclico (Directed Acyclic Graph)
PSHC	Parallel Stochastic Hill Climber (Escalador Estocástico Paralelo)

SUMÁRIO

1 Introdução

Com o avanço da nanotecnologia, os sistemas digitais estão se tornando cada vez mais complexos e miniaturizados. À medida que nos aproximamos dos limites da física determinística e adentramos o domínio da mecânica quântica, a prevalência e o impacto de defeitos em dispositivos eletrônicos se tornam mais significativos e desafiadores de gerenciar. A transição para escalas menores introduz uma maior suscetibilidade a falhas, exigindo abordagens inovadoras para garantir a confiabilidade e a robustez dos circuitos digitais. Para explorar plenamente o potencial das nanotecnologias emergentes (CHEN et al., 2021) a síntese lógica, não deve apenas fornecer suporte, mas também atuar como facilitadora para o desenvolvimento dessas tecnologias (AMARÚ et al., 2015). Assim, a EDA, na era da nanoeletrônica, deve ser capaz de gerar designs digitais que sejam tolerantes a defeitos de fabricação e cientes das variabilidades inerentes aos processos de manufatura. Curiosamente, sistemas biológicos moldados pela evolução exibem características de tolerância a defeitos e sensibilidade à variabilidade, adquiridas por meio de degeneração e redundância (EDELMAN; GALLY, 2001). Dessa forma, a CGP tem se destacado como um método promissor para a evolução de sistemas digitais tolerantes a defeitos. Ademais, estratégias naturais baseadas na evolução e voltadas para a gestão de defeitos podem ser reincorporadas no contexto de circuitos digitais evolutivos, especialmente quando se evita a otimização em direção a genótipos mínimos, o que pode ser alcançado ao promover a evolução na presença de defeitos (MILANO; NOLFI, 2016; MILANO; PAGLIUCA; NOLFI, 2019). Circuitos aritméticos básicos desempenham um papel crucial no desempenho e na eficiência energética de diversas tarefas computacionais. Um gerador de paridade ímpar é um circuito lógico combinacional simples, utilizado para determinar se um padrão de entrada de n bits contém um número ímpar de "uns". Geradores de paridade de dois bits (ou seja, portas XOR) são amplamente utilizados em diversos circuitos aritméticos, incluindo somadores. Este estudo relata a melhoria da tolerância a defeitos, alcançada por meio da geração de geradores de paridade que foram evoluídos na presença de falhas de componente, explorando os conceitos de redundância e degeneração.

1.1 Objetivos

O trabalho proposto tem como foco principal desenvolver circuitos lógicos robustos e tolerantes a falhas, utilizando técnicas avançadas de síntese e otimização, como a Programação Genética Cartesiana (CGP), para enfrentar os desafios impostos pela miniaturização e complexidade dos sistemas digitais em escala nanométrica. A seguir, detalhamos os objetivos gerais e específicos que guiarão a pesquisa.

1.1.1 Objetivo Geral

Explorar técnicas de Programação Genética Cartesiana a fim de sintetizar circuitos lógicos robustos, capazes de operar de forma confiável mesmo na presença de falhas de fabricação. Além de avaliar o impacto que as técnicas de redundância e entrelaçamento causam na robustez dos circuitos, e como a evolução pode se relacionar com essas técnicas.

1.1.2 Objetivos Específicos

- **Investigar o Impacto de Falhas de Fabricação em Circuitos Lógicos:** Analisar como defeitos em componentes, como portas lógicas defeituosas (Stuck-At Faults), afetam o funcionamento de circuitos combinacionais. Estudar a propagação de erros em circuitos e identificar pontos críticos onde falhas podem comprometer a funcionalidade do sistema.

- **Explorar Técnicas de Tolerância a Falhas:** Avaliar métodos tradicionais de tolerância a falhas, como a Redundância Modular Tripla (TMR), e suas limitações em contextos de alta complexidade e miniaturização. Investigar estratégias alternativas, como a degeneração e a redundância funcional, inspiradas em sistemas biológicos.

- **Aplicar a Programação Genética Cartesiana (CGP) para Síntese de Circuitos Robustos:** Utilizar a CGP para evoluir circuitos lógicos que sejam tolerantes a falhas, simulando cenários com componentes defeituosos durante o processo de evolução. Explorar a capacidade da CGP de gerar soluções não óbvias e altamente adaptativas, que incorporem redundância e degeneração de forma natural.

- **Implementar e Validar Circuitos Combinacionais Básicos:** Focar em circuitos combinacional fundamentais, como geradores de paridade ímpar, devido à sua

importância em tarefas computacionais. Validar os circuitos evoluídos em cenários práticos, comparando seu desempenho com soluções convencionais.

- **Analisar a Escalabilidade e Aplicabilidade das Soluções Propostas:** Investigar a viabilidade de aplicar as técnicas desenvolvidas em circuitos mais complexos, Avaliar o custo computacional e a eficiência do processo de evolução, buscando otimizar o tempo e os recursos necessários.

2 Revisão da Literatura

2.1 Tolerância a Falha

A tolerância a falhas em sistemas computacionais é uma área que emergiu em paralelo com o desenvolvimento dos computadores modernos. Um dos primeiros marcos nesse campo foi o trabalho de John Von Neumann, que, ao projetar a primeira máquina de programa armazenado, introduziu o conceito de sintetizar computadores confiáveis a partir de componentes não confiáveis. Ele propôs as ideias de redundância e replicação, fundamentais para muitas das arquiteturas de sistemas contemporâneos.

O programa espacial da década de 1960 deu grande impulso à tolerância a falhas, impulsionado pela necessidade de construir sistemas que funcionassem por longos períodos sem manutenção. As missões tripuladas foram um catalisador importante para o desenvolvimento de técnicas de confiabilidade, e a demanda por sistemas robustos e resilientes continuou a crescer, impulsionando o desenvolvimento de metodologias avançadas para garantir a disponibilidade e a eficiência dos sistemas (GREEN et al., 2019).

O avanço das técnicas de integração em larga escala (VLSI, Very Large Scale Integration) também desempenhou um papel essencial, tornando a replicação e a redundância mais acessíveis e economicamente viáveis. Com isso, a tolerância a falhas deixou de ser um tema restrito a áreas específicas como defesa, telecomunicações e programas espaciais, e passou a permear diversos setores da sociedade moderna, incluindo ciências, negócios e indústrias criativas. Atualmente, a tolerância a falhas se tornou um campo disciplinar amplo, englobando diversas abordagens que cobrem tanto o hardware quanto o software. No hardware, técnicas como redundância física e de informação, além do desenvolvimento de circuitos autoverificáveis, têm sido amplamente exploradas. Essa área de pesquisa é impulsionada por três fatores principais: a necessidade de alta confiabilidade, a necessidade de alta disponibilidade e o impacto direto da perda de confiabilidade no desempenho do sistema. O foco nesses três aspectos é vital, principalmente em sistemas críticos, como os de telecomunicações, onde o tempo de inatividade é extremamente custoso.

2.2 Técnicas de Tolerância a Falhas

A principal técnica de tolerância a falhas é a NMR (N-modular Redundancy), mais especificamente a TMR (Triple Modular Redundancy) (BURLYAEV, 2015). A técnica é inspirada no Multiplexing proposto por Von Neumann nos anos 1950 onde três cópias idênticas de um circuito são implementadas, e um mecanismo de votação de maioria decide a saída correta com base nos resultados dessas três cópias. Essa técnica permite mascarar falhas em qualquer uma das três réplicas, desde que apenas uma falhe. A vantagem principal da TMR em circuitos digitais combinacionais é sua capacidade de mascarar falhas em tempo real, garantindo a continuidade da operação sem necessidade de intervenção imediata ou reconfiguração.

Uma limitação importante a ser considerada na NMR, é a sensibilidade do sistema final a falhas no sistema de votação. A falha só pode ocorrer em um dos módulos replicados, uma vez que se ocorrer no sistema de votação pode gerar uma saída contrária, ou até mesmo, a depender da modelagem de falha utilizada, uma saída independente dos módulos. Por exemplo, um erro no sistema de votação que defina a saída como 0 em todos os cenários.

2.3 Modelo de Falhas

O termo **falha** é utilizado para identificar o evento físico inicial que pode comprometer o funcionamento de um sistema , enquanto o termo **erro** refere-se ao estado incorreto ou indesejado do sistema resultante dessa falha. A maneira como modelamos as falhas e suas consequências é definida por um **modelo de falha** (PIERCE, 1964). Um **fracasso** ocorre quando o serviço fornecido pelo sistema se desvia do comportamento esperado ou correto, afetando a lógica e a integridade do processamento.

Nesse contexto, a tolerância a falhas é a capacidade de um sistema lógico computacional de evitar fracassos mesmo na presença de falhas, garantindo assim a entrega do serviço especificado e resultados corretos. A correção de um processo computacional é determinada pela ausência de saídas incorretas, o que é fundamental para a confiabilidade de sistemas que dependem de lógica precisa e resultados exatos.

Em muitos cenários de estudo da microeletrônica, especialmente para simplificar a análise, considera-se que a falha de um componente é um evento independente da falha

de outros componentes. No modelo de falhas que será adotado, cada porta lógica em um circuito tem uma chance constante e independente de falhar. Ou seja, a probabilidade de uma porta falhar é fixa, e a falha de uma porta não afeta a probabilidade de falhas nas outras portas.

A falha de uma porta terá como resultado uma saída 0 ou 1, simbolizando uma porta desligada ou sempre ativa. Este modelo é conhecido como Stuck-At Fault (SAF) (UBAR et al., 2024), e é uma abordagem comum para análise de robustez em sistemas lógicos.

Com essas propriedades, podemos assumir que a chance de falha de componentes do modelo seguirá uma distribuição binomial, o que permite calcular a probabilidade de um número específico de falhas em um conjunto de n portas lógicas. A equação que usamos para determinar a probabilidade de exatamente x falhas em n portas é definida como em 2.1:

$$P(X = x) = \binom{n}{x} p^x (1-p)^{n-x} \tag{2.1}$$

Onde:

- n é o número total de portas lógicas no circuito.
- p é a probabilidade de falha de uma porta.
- x é o número de falhas que queremos calcular.

É importante mencionar que essa modelagem tem consequências na generalização da análise, já que não serão consideradas falhas que sejam provocadas de forma generalizada ou correlacionada entre os componentes, como aquelas provocadas por interferências eletromagnéticas ou surtos elétricos. No entanto, a escolha do modelo de falha foi importante para que possamos simplificar o cálculo da probabilidade de falhas de componentes em mais de 1 porta (DJUPDAL; HADDOW, 2011), como iremos demonstrar mais à frente, e com isso o custo computacional é consideravelmente reduzido.

2.4 Programação Genética Cartesiana

A crescente complexidade dos circuitos microeletrônicos e o alto custo de prototipagem tornaram técnicas de modelagem, simulação e otimização extremamente populares.

Nesse contexto, muito avanço já foi feito, possibilitando a simulação utilizando ferramentas de EDA para cobrir processos, dispositivos e ferramentas que permitem a exploração de um vasto espaço de design e otimização, facilitando o desenvolvimento de soluções inovadoras que, muitas vezes, poderiam não ser intuitivas para projetistas humanos.

Usar um computador para modelar circuitos torna-se especialmente vantajoso à medida que a complexidade do circuito aumenta, permitindo uma exploração eficiente de uma vasta possibilidade de design e espaço de otimização. Além disso, a capacidade de um computador gerar conexões não óbvias, que inicialmente podem parecer pouco promissoras para um ser humano, pode levar a soluções inovadoras e altamente eficazes. Empregando o conceito de "sobrevivência do mais apto", ou seja, do melhor fitness, a Programação Genética (GP) é uma técnica de computação evolutiva na qual programas de computador evoluem através de mutações e seleções ocorrendo de forma probabilística, em vez de seguir um padrão determinístico fixo. No presente estudo, abordamos o desafio da modelagem utilizando uma implementação do CGP, uma variante da programação genética (GP) que representa soluções como DAGs (Grafos Dirigidos Acíclicos), organizados em uma grade cartesiana (MANAZIR; RAZA, 2019), nesse cenário, haveriam conexões entre as colunas (apenas as imediatamente anteriores a ela). Na implementação do projeto, não houve limitação nas conexões das colunas da grade cartesiana de forma que as conexões possam ser feitas com qualquer coluna que seja anterior a ela. A Figura 1 representa um circuito multiplexador 4x1 através de um DAG que foi evoluído através da implementação de CGP em trabalhos anteriores (SOARES; TAVORA; BLAWID, 2024) que participei.

Um dos principais desafios da CGP é a avaliação de *fitness*, que é a parte mais demorada do ciclo evolutivo. A função de *fitness*, neste caso, mede a adequação de uma solução comparando o número de saídas corretas geradas pelo circuito com o número total de saídas possíveis. Essa métrica é crucial para guiar a evolução, permitindo que o algoritmo evolutivo elimine soluções ineficazes e retenha as mais promissoras. Os circuitos projetados neste estudo foram construídos usando exclusivamente portas NAND, um padrão comum em circuitos digitais, já que se trata de uma porta universal, o que permite expressar qualquer outra função lógica a custo de uma área maior.

A CGP foi responsável por gerar uma ampla variedade de circuitos combinacionais, todos representados como grafos acíclicos direcionados, onde os nós representam portas NAND e as arestas conectam entradas e saídas de forma hierárquica. A estrutura dos

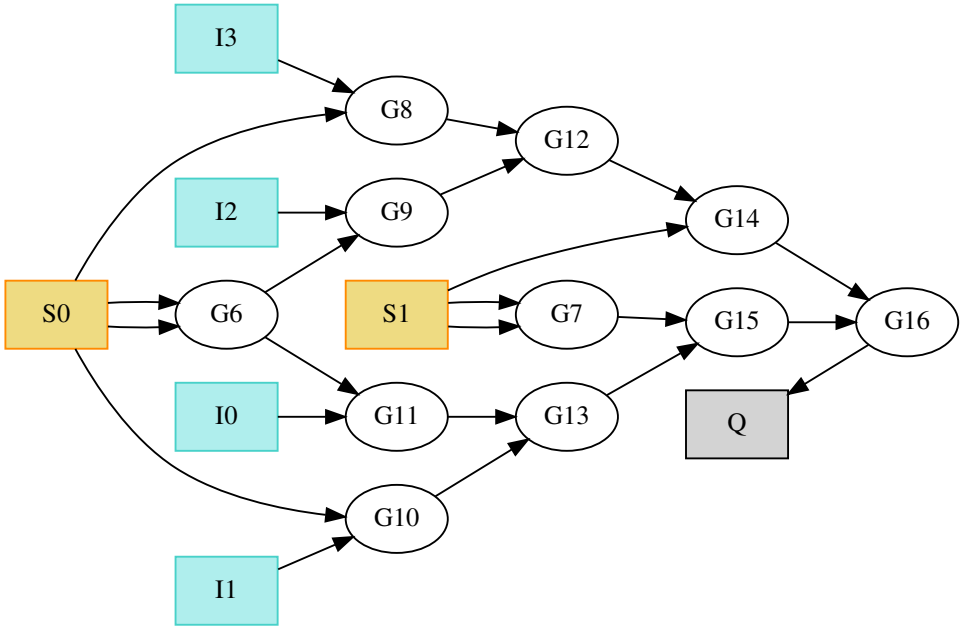

Figura 1 – Multiplexador 4x1 representado por um DAG e otimizado pelo algoritmo desenvolvido.

grafos facilita a otimização, permitindo que a CGP explore combinações inusitadas de portas lógicas, resultando em soluções inovadoras.

3 Fundamentação Teórica

3.1 Representação

Como mencionado nos capítulos anteriores, foi utilizado o CGP como paradigma evolutivo. Com ele, podemos utilizar uma representação cartesiana de DAGs como podemos ver na Figura 2 que representa o DAG da Figura 1.

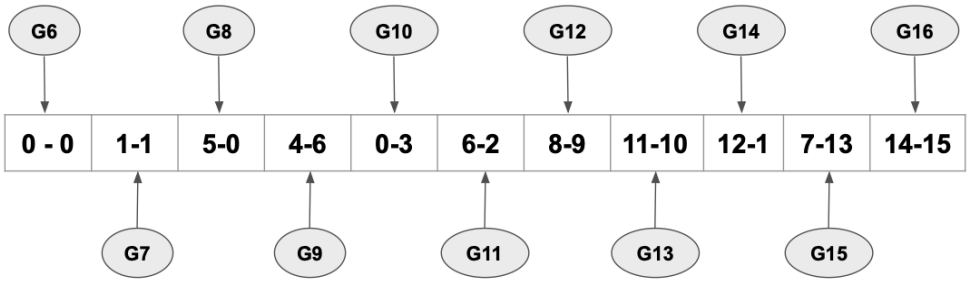

Figura 2 – Expressão cartesiana do Genótipo do MUX 4x1.

Essa representação permite que possamos utilizar uma linguagem de alto nível como *Python* para realizar operações típicas de algoritmos genéticos com uma performance computacional aceitável.

Cada elemento do vetor da Figura 2 representa um **Gene**, a sequência completa dos genes representa um **Genótipo**. A lógica resultante das conexões de cada gene é o resultado do circuito (*output*), que é representado pelo **Fenótipo** do indivíduo (RYSER-WELCH; MILLER, 2016). O conjunto destas características (Gene, Genótipo, Fenótipo) é o que compõe o **Genoma**. No que se refere à simbologia dos circuitos lógicos, o Gene representa uma porta, mais especificamente uma NAND de duas portas, já que utilizaremos apenas ela neste trabalho. A numeração contida no Gene vai representar as conexões que a porta possui, ou seja, o gene G6 representa uma NAND sendo aplicada à entrada. Desta forma, alterar a numeração de um gene é equivalente a alterar a conexão da porta correspondente, alterando a arquitetura do circuito e permitindo que novas expressões lógicas sejam descobertas. A esta operação, chamaremos de **Mutação**. Durante a etapa de mutação, é preciso garantir que o índice seja modificado para um valor menor que o índice da porta em questão. Por exemplo, a porta G10 ao sofrer mutação, só pode se conectar com portas de índices anteriores ou entradas (G9, G8, ou S0, S1...). Isso garante

que o circuito permaneça combinacional ao final da mutação, e não seja alterado para uma lógica sequencial.

Com isso, através da representação cartesiana, podemos utilizar uma linguagem de alto nível como *Python* para realizar as operações de mutação ao longo das evoluções a um custo operacional baixo, sem a necessidade de utilizar ferramentas de software especializadas em design de circuitos lógicos, além de reduzir o custo de automação da ferramenta.

3.2 Estrutura do Genótipo

Ao definirmos o tamanho do genótipo, estamos estabelecendo o espaço de probabilidade que o nosso genoma tem para encontrar a solução desejada. Aumentar ou diminuir este tamanho permite que o circuito aumente ou diminua a quantidade de soluções possíveis.

O circuito útil resultante é aquele que de fato impacta no *output* do sistema, ou seja, todos os genes que estão ligados (a qualquer distância) à saída. A esta parte, chamaremos de **Genes Ativos**, a parte que não está conectada de forma alguma à saída, chamaremos de **Genes Mortos**.

Essas duas estruturas foram cruciais para o desempenho da ferramenta. Uma vez que os Genes Mortos não impactam na saída do sistema, podemos ignorá-los e não precisamos calcular as NANDs desses genes durante o cálculo do *fitness*, com isso, reduzimos o custo deste cálculo de N_{genes} para $N_{\text{genesAtivos}}$ (RYSER-WELCH; MILLER, 2016). Entraremos em mais detalhes sobre a performance mais à frente.

É importante mencionar que essa divisão da estrutura do genoma traz uma consequência importante para a evolução. O ponto é que um genoma pode ter vários genes inativos no seu genótipo, mas durante a evolução e a etapa de mutação, esses podem ser ativos, permitindo que o circuito altere a sua arquitetura e ative uma parte totalmente nova. Imaginemos o cenário abaixo, onde temos o circuito que representa uma porta XOR (Figura 3).

É perceptível que a porta 04 não se relaciona com a saída e, por consequência, é um Gene Morto. No entanto, durante a evolução, a porta 07 pode sofrer Mutação e

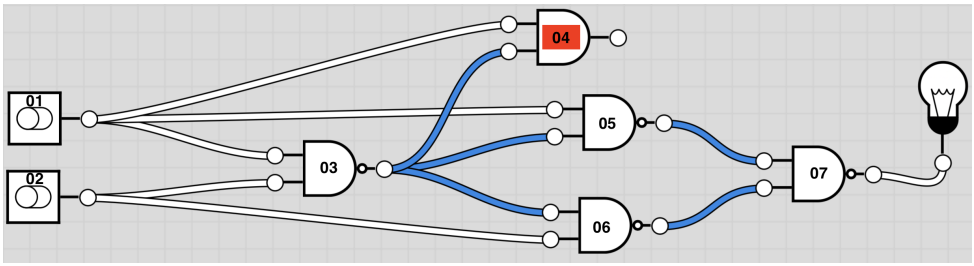

Figura 3 – Circuito XOR com uma NAND Morta.

alterar as suas ligações de 05-06 para 04-06. Isso ocasionaria na inativação da porta 05 e ativação da porta 04 como pode-se conferir na Figura 4:

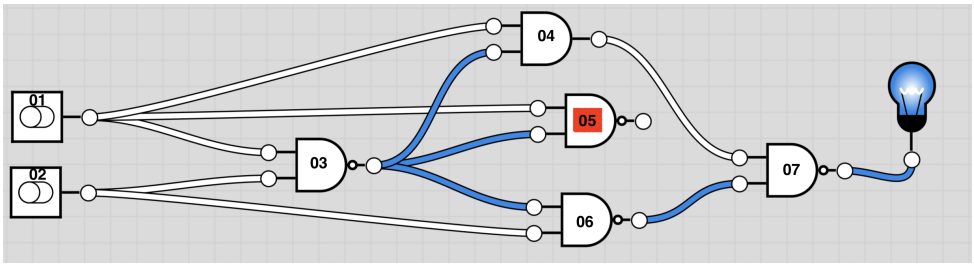

Figura 4 – Circuito XOR com ativação de NAND Morta.

Essa capacidade de alteração rápida da estrutura do circuito é o que torna a utilização do algoritmo evolutivo tão interessante e capaz de encontrar soluções inovadoras.

3.3 Estratégia Evolutiva

A base da implementação de CGP utilizada foi a estratégia $(1+\lambda)$, onde um único pai é escolhido a cada geração e produz λ filhos. Então, é selecionado o melhor genoma para ser o pai da próxima geração. Nesta estratégia, o pai é considerado em cada geração na escolha do melhor genoma; isso tende a garantir que a evolução não mude o *fitness* demasiadamente e se mantenha em direção a uma melhora. Também é importante notar que em cada geração há apenas 1 sobrevivente; todos os demais são eliminados. Isso é necessário devido ao custo computacional de gerenciar vários genomas em casos mais complexos.

O cálculo do *fitness* é a etapa mais complexa e custosa computacionalmente da evolução. Além disso, é onde a evolução passa mais tempo, já que em cada geração haverá pelo menos $(1+\lambda)$ cálculos de *fitness*.

Esta estratégia é bastante comum em aplicações de CGP a circuitos lógicos (MILANO; PAGLIUCA; NOLFI, 2019), e tende a apresentar os melhores resultados por per-

mitir uma variabilidade de soluções de maneira controlada. Na evolução, apenas a etapa de mutação é utilizada, já que técnicas de *Crossover* não se demonstraram úteis em cenários como este (RYSER-WELCH; MILLER, 2016). O algoritmo genético desenvolvido pode ser visto em *Algorithm.* 1.

Algorithm 1 Evolução do Genoma

Require: $logicFunction$
1: $bestParent \leftarrow$ novo Genoma($genome.numberOfGenes$, $genome.nInputs$, $genome.nOutputs$)
2: **while** true **do**
3: Incrementar $totalGeneration$
4: Limpar $listGenomes$
5: Calcular *fitness* de $bestParent$ com $logicFunction$ + Ruido
6: Adicionar $bestParent$ a $listGenomes$
7: **for** $i = 0$ até λ **do**
8: $child \leftarrow$ novo Genoma($bestParent.numberOfGenes$, $bestParent.nInputs$, $bestParent.nOutputs$)
9: Mutação de $bestParent$ e cópia dos genes para $child$
10: Calcular *fitness* de $child$ com $logicFunction$ + Ruido
11: Adicionar $child$ a $listGenomes$
12: **end for**
13: Copiar melhor genoma de $listGenomes$ para $bestParent$
14: **if** $totalGeneration \geq maxGeneration$ **then**
15: Encerrar evolução
16: **end if**
17: **end while**
18: **return** $bestParent$

Uma particularidade da implementação com relação à sua implementação básica é utilizado um fator estocástico simbolizando o ruído. Este ruído é importante para permitir que a evolução não fique presa em máximos locais, de forma que circuitos com o mesmo *fitness* real, mas com ruídos diferentes, vão variar e se permitir encontrar soluções mais promissoras em algumas gerações à frente (MILANO; PAGLIUCA; NOLFI, 2019).

Existem outras estratégias evolutivas comuns em relação aos algoritmos evolutivos, como por exemplo a estratégia $(\mu + 1)$. Nela, um conjunto de μ pais gera 1 descendente. A seleção é feita entre os pais e os descendentes, escolhendo os μ melhores indivíduos para a próxima geração. Essa abordagem é amplamente utilizada na evolução de redes neurais, uma vez que tende à evolução tende a selecionar indivíduos localizados em regiões de alto *fitness* do espaço genético, mas que são caracterizadas por baixa variabilidade fenotípica e, consequentemente, baixa capacidade de evolução. Para o caso da evolução de redes

neurais, isso pode não ser um problema, já que, no geral, o objetivo é principalmente maximizar a acurácia (ou outra métrica especificada), mas para a solução que estamos trabalhando, a variabilidade genética e capacidade de explorar soluções menos óbvias é o principal objetivo.

Outra estratégia levada em consideração foi a PSHC (MILANO; PAGLIUCA; NOLFI, 2019). Nela, cada pai é adaptado por um número fixo de variações (mutações), gerando candidatos com mutações. O melhor candidato obtido durante a fase de variação substitui o pai original, mas de forma a reduzir a competição direta entre indivíduos, permitindo que cada um evolua de forma semi-independente. Funciona como um algoritmo de ilhas, onde cada indivíduo é uma 'ilha' que evolui separadamente, com interações ocasionais. Aplicando ao problema proposto, o PSHC, operando com múltiplos indivíduos evoluindo em paralelo, pode introduzir uma variabilidade excessiva, dificultando a convergência para soluções ótimas. Além disso, o PSHC, por outro lado, requer o gerenciamento de múltiplos indivíduos e suas variações, o que pode se tornar custoso em termos de tempo e recursos, especialmente em problemas com alto custo de avaliação de *fitness*.

3.4 Cálculo de Fitness

O ponto mais importante durante a evolução é o cálculo do *fitness* do genoma. É nele que definiremos a função lógica alvo que desejamos encontrar ao final da evolução, e é esta etapa que tem o maior impacto na performance de tempo do algoritmo. O maior desafio é utilizar o cálculo de *fitness* de uma maneira eficiente o suficiente de forma que o algoritmo evolutivo leve um tempo aceitável para encontrar a melhor solução. Isso se deve à quantidade gigantesca de cálculos de *fitness* que serão feitos ao longo de toda a evolução.

Uma estratégia direta seria aplicar a lógica de cada porta (representada por cada gene) em todo o circuito. Desta forma, podemos ter uma complexidade $O(n)$, onde n é o tamanho do genótipo. Analisando este cenário, para a estratégia evolutiva apresentada, temos um cenário em que o *loop* principal é executado até que uma condição de parada seja atingida ($maxGeneration$), desta forma, o *loop* principal é executado por G gerações. Já para cada geração, o cálculo de *fitness* de $bestParent$ é $O(n)$, e o *loop* interno é executado λ vezes, onde em cada iteração envolve operações $O(n)$ (criação, mutação e cálculo de

fitness). Por fim, a seleção do melhor genoma é $O(\lambda)$. Isso nos dá uma complexidade final do algoritmo evolutivo de $O(Gn\lambda)$. Dado que o valor máximo de gerações permitidas é definido de forma a permitir uma quantidade de gerações na ordem de milhares a milhões, estamos falando de um cenário pior do que uma complexidade $O(n^2)$.

Desta forma, é preciso utilizar uma estratégia mais otimizada para o cálculo do *fitness*. O primeiro passo é estabelecer o cálculo das portas lógicas de um circuito apenas para aquelas que de fato afetam a saída do sistema, ou seja, **Genes Ativos**. Como visto em (RYSER-WELCH; MILLER, 2016), a maior parte do genoma é composta de **Genes Mortos**, logo o custo do cálculo do *fitness* se resume a uma fração do seu tamanho real (de N_{genes} para $N_{\text{genesAtivos}}$). A implementação base do cálculo de *fitness* pode ser vista em *Algorithm*. 2 e 3.

Algorithm 2 Cálculo Fitness Real

Require: Função lógica *logicFunction*
Ensure: Fitness do genoma
1: Identificar genes inativos (*self.identify_ deadGenes*)
2: Inicializar contador de fitness (*fitnessCounter* $\leftarrow 0$)
3: **for** cada combinação de entrada na tabela verdade **do**
4: Avaliar saida de cada porta *NAND* ativa
5: Comparar saída do genoma com saída esperada (*logicFunction*)
6: **if** saídas coincidem **then**
7: Incrementar *fitnessCounter*
8: **end if**
9: **end for**
10: Calcular fitness final (*self.fitness* $\leftarrow$ *fitnessCounter* / *self.possiblesOutputs*)

Algorithm 3 Cálculo de Fitness com Ruído

Ensure: Fitness com ruído
1: Gerar ruído aleatório (*noise* $\leftarrow$ *random.uniform*)
2: Adicionar ruído ao fitness (*self.noiseFitness* $\leftarrow$ *self.fitness* + *noise*)

Como pode ser visto em *Algorithm*. 2 e 3, o cálculo do *fitness* não foi realizado em um ambiente externo de simulação e teste de circuitos lógicos. Os cálculos são feitos diretamente em *Python*, no mesmo ambiente da implementação do próprio algoritmo evolutivo. Esta escolha foi feita por dois principais motivos. Como mencionado, o cálculo do *fitness* é a etapa mais custosa da evolução. Utilizar uma simulação externa acrescentaria um gargalo de acionamento de interface entre as ferramentas, além da limitação da velocidade de um software mais complexo como *Quartus* II que é consideravelmente menos

performático do que uma linguagem de alto nível como *Python*. Além disso, manter o todas da evolucao no mesmo ambiente, facilita a integração entre as suas etapas, onde é mais fácil realizar as operações genéticas numa linguagem de alto nível.

4 Metodologia

A principal proposta deste trabalho é a utilização de simulação de falhas de componentes durante a evolução dos genomas. Para isto, utilizou-se o *Algorithm.* 4 e 5 para o cálculo do *fitness*.

Algorithm 4 Cálculo de Fitness com Falha injetada

Require: Função lógica *logicFunction*
Ensure: Fitness final com injeção de falhas
1: TruthFitness ← Cálculo Fitness Real
2: Obter genes ativos
3: Calcular distância média dos genes (*avgDistance*)
4: **for** cada gene ativo **do**
5: **if** distância do gene ≥ *avgDistance* **then**
6: Calcular fitness com falha no gene (*faultFitness*)
7: Acumular fitness com falha (*totalFaultFitness*)
8: Acumular n
9: **end if**
10: **end for**
11: Calcular fitness final (*self.fitness* ← média de *TruthFitness* e $totalFaultFitness/n$)

Algorithm 5 Iteração de Cálculo de Fitness com Falha Injetada

Require: Função lógica *logicFunction*, índice de falha *indexFault*
Ensure: Fitness com injeção de falha
1: Identificar genes inativos (*self.identify_ deadGenes*)
2: Inicializar contador de fitness (*fitnessCounter* ← 0)
3: Gerar tabela verdade (*TrueTable* ← *self.getCartesianProduct(l)*)
4: **for** cada combinação de entrada na tabela verdade **do**
5: Avaliar saida de cada porta NAND ativa com falha no gene *indexFault*
6: Comparar saída do genoma com saída esperada (*logicFunction*)
7: **if** saídas coincidem **then**
8: Incrementar *fitnessCounter*
9: **end if**
10: **end for**
11: Retornar fitness com falha (*fitnessCounter* / *self.possiblesOutputs*)

A presença do índice da porta no cálculo do *fitness* é utilizada para garantir que a falha não seja injetada em qualquer porta. Como visto em (PIERCE, 1964), as falhas nas portas impactam de maneira diferente de acordo com a distância que a porta está do *output* do sistema. De maneira geral, é preciso garantir que o circuito tenha espaço suficiente para compensar o erro na porta em questão. O caso mais extremo seria se a

falha ocorresse na porta que representa a saída do sistema, onde, desta forma, teríamos um circuito que não é uma função de suas entradas.

Como mencionado, durante o cálculo do *fitness* são apenas as portas ativas que são calculadas, além disso, é utilizada a distância de cada porta para decisão de injeção de falha. No *Algorithm.* 6 pode-se verificar como foi implementada essa função.

Algorithm 6 Obter genes ativos

```
1: ToEvaluate ← [False] × (numeroDeGenes - nSaidas) + [True] × nSaidas
2: Distancias ← [∞] × numeroDeGenes
3: Distancias[-1] ← 0
4: p ← numeroDeGenes - 1
5: while p ≥ 0 do
6:    if ToEvaluate[p] then
7:       entradas ← genotipo[p].split(-)
8:       entrada1 ← int(entradas[0])
9:       entrada2 ← int(entradas[1])
10:      x ← entrada1 - nEntradas
11:      y ← entrada2 - nEntradas
12:      if x ≥ 0 then
13:         ToEvaluate[x] ← True
14:         Distancias[x] ← min(Distancias[x], Distancias[p] + 1)
15:      end if
16:      if y ≥ 0 then
17:         ToEvaluate[y] ← True
18:         Distancias[y] ← min(Distancias[y], Distancias[p] + 1)
19:      end if
20:   end if
21:   p ← p - 1
22: end while
23: DistanciasFinais ← [(i + nEntradas, v) para i, v em Distancias se v ≠ ∞]
24: Fim Função
```

É importante notar que com a abordagem proposta, o cálculo do *fitness* se mostra ainda mais custoso computacionalmente falando, já que iremos iterar sobre ele várias vezes.

4.0.1 Proposições Assumidas

Por esses e outros motivos, com base no algoritmo evolutivo apresentado em *Algorithm.* 1 e na estrutura cartesiana apresentada na Figura 2, alguns pressupostos foram necessários.

1. As numerações de identificação começam em zero, e as primeiras N_{inputs} numerações são reservados para as entradas do sistema. Por exemplo, o MUX_{4x1} representado nas Figuras 2 e 1 possui 6 números reservados. Sendo 2 números para o seletor de 2 bits (0 e 1) e 4 números para a entrada de 4 bits (2,3,4 e 5), desta forma, a numeração das portas começa a partir de 6 (G6, G7, G8...). Esta numeração é intrínseca ao circuito, e pode variar a depender do seu tipo. Por exemplo, um gerador de paridade impar de 4 bits só teria 4 números reservados para representar as suas entradas, então teria as portas sendo numeradas a partir de 4 (G4, G5...).

2. As ultimas N_{outputs} numerações do do circuito são reservadas para a saída do sistema. Por exemplo, no mesmo cenário mencionado anteriormente, como para o MUX_{4x1} temos apenas 1 bit de saída, a ultima porta simboliza o *output* (G16). Semelhante à propriedade anterior, esta varia a depender do circuito. Por exemplo, um Somador de 1 bit possui uma saída de 2 bits devido a presença do possível *overflow*. Logo, as ultimas 2 portas seriam a representação do seu *output*.

3. As falhas de componente seguiram o padrão SAF (*Stuck-At Fault*) sendo simbolizadas por saídas sempre 0 ou sempre 1.

4. Como mencionado na sessão do modelo de falhas, vamos assumir uma distribuição de probabilidade binomial para os casos de portas que falham.

4.0.2 Propriedades

Assumindo que as proposições acima são verdadeiras, temos as seguintes propriedades:

1. Para um circuito de tamanho N_{genes}, a quantidade de números necessários para representar este genótipo é expressa por:

$$N_{\text{indices}} = N_{\text{genes}} + N_{\text{inputs}} - 1 \tag{4.1}$$

2. Uma gene só poderá sofrer mutação para genes de índice menor que ele. Isso garantirá que o circuito final permaneça combinacional.

3. Durante o calculo do *fitness* as falhas serão injetadas apenas nas portas que possuem uma distancia maior ou igual à média das distancias.

4. O *fitness* final é composto por um componente na ausência falhas e um componente definido como a média do *fitness* na presença de uma falha em cada porta (respeitando a propriedade 3).

Por conta da proposição 4, podemos calcular o *fitness* do genoma apenas para os casos onde há a presença de apenas uma falha de componente, já que os casos de mais de uma falha possuem uma probabilidade desprezível. Isso permite que o cálculo do *fitness* simule as falhas para apenas uma porta, o que reduz o custo computacional.

A escolha de hiperparâmetros para evolução é fundamental para o bom funcionamento do algoritmo. Com base no trabalho feito em (SOARES; TAVORA; BLAWID, 2024), temos uma amostra de valores de hiperparâmetros que tiveram boa performance para evolução de multiplexadores, que são de complexidade semelhante a um gerador de paridade. A escolha foi semelhante aos melhores resultados no artigo mencionado, onde a taxa de mutação escolhida foi de 10% e o λ foi definido como 10. O fator estocástico é definido em 0.03, o que é baixo, mas suficiente para minimizar os efeitos de máximos locais durante a evolução.

5 Experimentos e Análise de Resultados

O algoritmo desenvolvido e todos os genomas evoluídos podem ser vistos no repositório do GitHub Logic-Circuits-Evolution. Através dele, uma série de evoluções foi realizada. A performance do algoritmo foi eficiente, sendo capaz de encontrar um genoma de *fitness* máximo em uma média de 4 minutos de execução para uma Função Alvo como um gerador de paridade ímpar de 4 bits.

Podemos pegar de exemplo uma das soluções (como a vista na Figura 6) e comparar com a solução mínima do estado da arte expressa na Figura 5, e comparar a robustez dos dois genomas.

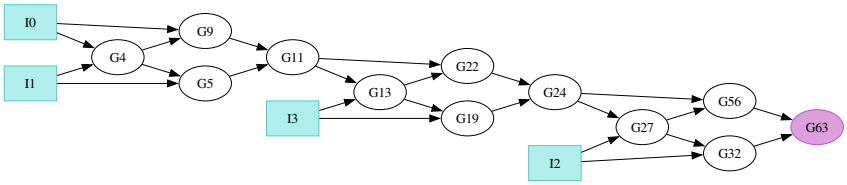

Figura 5 – Genoma de solução mínima (Gerador de Paridade Ímpar).

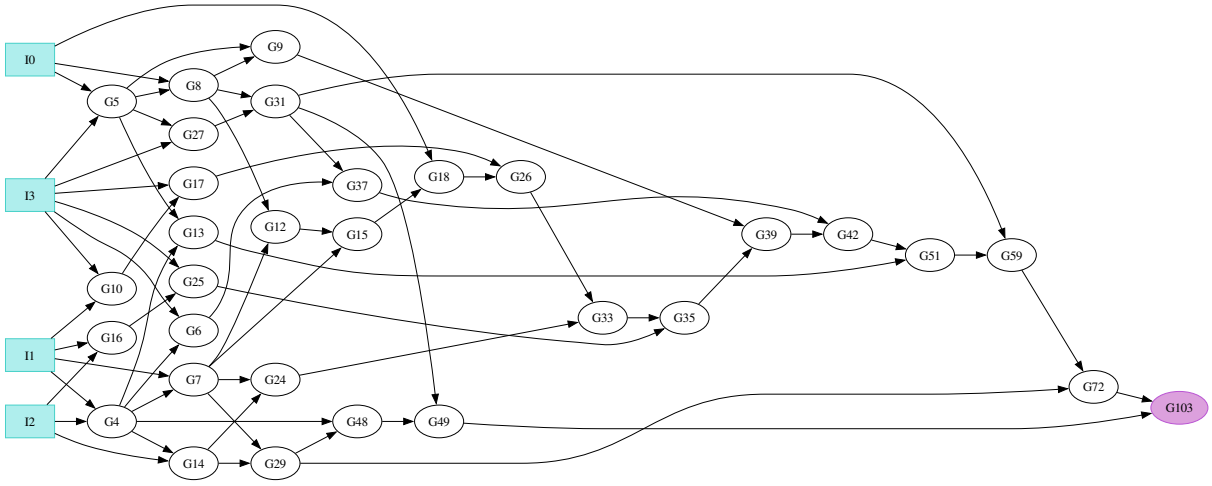

Figura 6 – Genoma evoluído até um *fitness* máximo (Gerador de Paridade Ímpar).

Inicialmente, vamos analisar a robustez em um cenário sintético, onde a falha de componente será injetada em cada porta, e desta vez, não limitaremos a falha para garantir um cenário o mais geral possível. Esta comparação pode ser vista na Tabela 1

O aumento do *fitness* da solução proposta é uma melhoria bastante expressiva na robustez. É interessante analisar a capacidade do circuito de compensar as falhas nas portas de forma a manter a sua funcionalidade. No entanto, é perceptível o aumento no

Tabela 1 – Cenário Binomial: Robustez Contra Falha Única do Genoma Evoluído e da Solução Mínima do Estado da Arte.

Genoma Evol. (%)	**Solução Min. (%)**	**Dif (%)**
90.7	71.8	+18.9

tamanho do circuito, onde a solução evoluída precisa de 31 portas, enquanto a solução mínima tem apenas 12 portas. É importante analisar que a diferença de tamanho também impacta diretamente na probabilidade de falha única em cada abordagem. Como pode-se ver na Tabela 2, a partir da Equação 2.1, pode-se observar que em todas as chances de falha (1% a 5%) a chance de falha única na solução mínima é expressivamente menor.

Tabela 2 – Análise da distribuição de probabilidade de uma porta falhar (P(X = 1)) do genoma evoluído contra a solução mínima

p = Chance de Falha (%)	**Genoma Evol. (%)**	**Solução Min. (%)**
1	23.4	10.7
2	34.2	19.2
3	37.3	25.7
4	36.1	30.6
5	32.6	34.1

Avaliando em um cenário estocástico, podemos simular uma falha com base em uma chance aleatória de cada porta. Logo, não temos mais uma das premissas para a distribuição binomial, o que nos impede de assumir a chance de 2 ou mais portas falharem como sendo desprezível. Desta forma, podemos observar um cenário geral de robustez destes dois circuitos. Esta análise pode ser vista na Tabela. 3, onde são exibidos os resultados de média de 100.000 cálculos de *fitness* para cada chance de falha entre 1% a 5% .

Tabela 3 – Cenário Estocástico: Robustez Contra Distribuição de Falhas do Genoma Evoluído e da Solução Mínima do Estado da Arte.

Chance de Falha (%)	**Genoma Evol. (%)**	**Solução Min. (%)**	**Dif (%)**
1	94.0	93.8	+0.2
2	89.1	88.7	+0.4
3	84.7	84.3	+0.4
4	80.9	80.2	+0.7
5	77.3	76.6	+0.7

Neste caso, a melhoria no *fitness*, apesar de constante, é bem menor. Estamos lidando com a relação entre o tamanho do circuito e a chance aleatória de falha em cada porta. A consequência dessa aleatoriedade é que a chance de um circuito possuir portas defeituosas cresce ao passo que o tamanho do circuito aumenta. Ou seja, este resultado, mesmo que positivo para o circuito evoluído, não necessariamente é indicativo da capacidade do circuito de solução mínima resistir a falhas, já que em muitos cenários, este circuito pode não ter nenhuma falha de componente.

Nas Figuras 7 e 8 podemos observar a distribuição do *fitness* ao longo dos 100.000 cálculos realizados no cenário de falha estocástica.

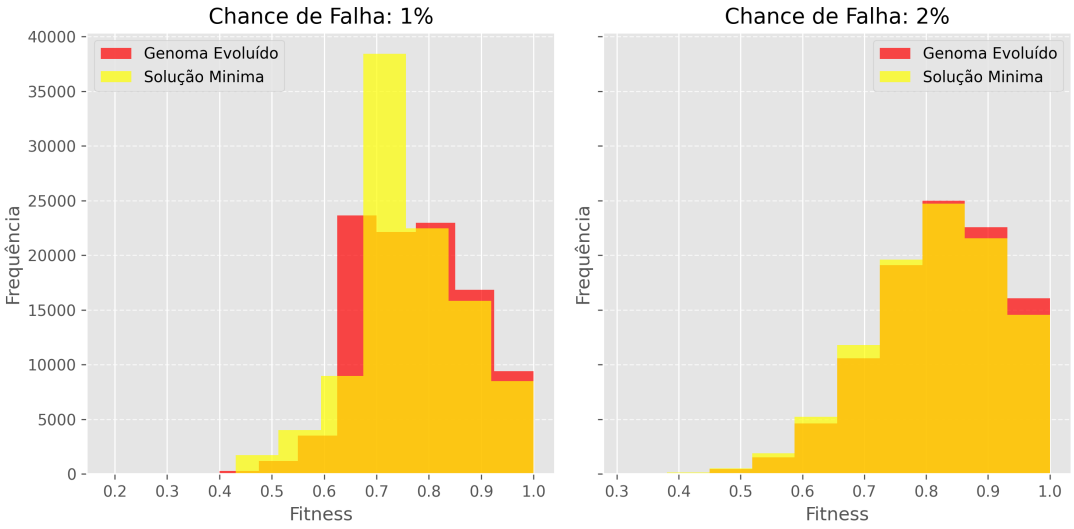

Figura 7 – Cenário Estocástico: Distribuição do *fitness* em barras (Parte 1).

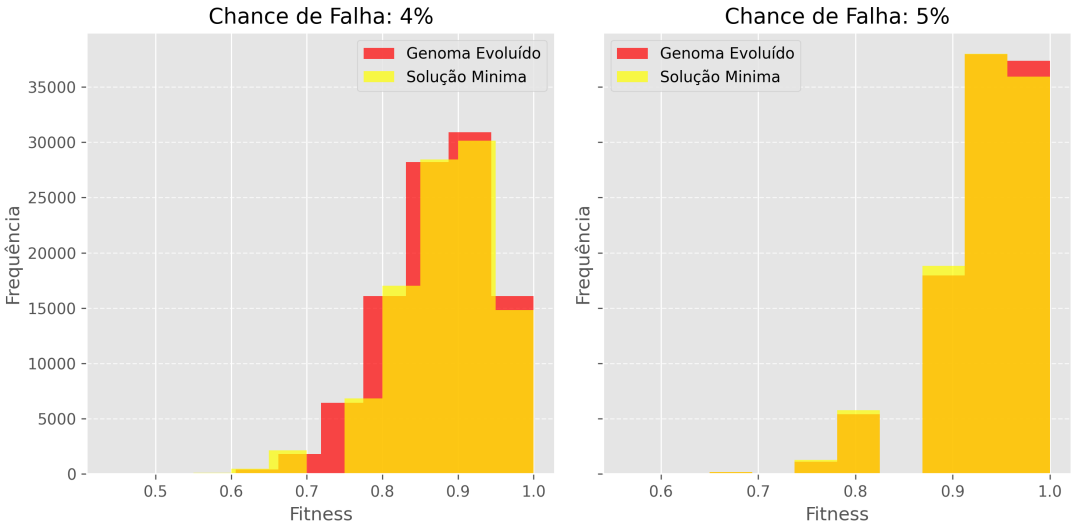

Figura 8 – Cenário Estocástico: Distribuição do *fitness* em barras (Parte 2).

É perceptível que a distribuição do *fitness* tende a se deslocar para a direita no caso do genoma evoluído. De maneira geral, em todos os cenários, a faixa de *fitness* máximo é maior para o caso do genoma evoluído, indicando que este tendeu a superar as falhas injetadas completamente em mais cenários. Além disso, a faixa de menor *fitness* é consistentemente maior nas barras da solução mínima, o que indica que em mais cenários, a solução mínima teve o *fitness* fortemente impactado pela falha injetada, enquanto que o genoma evoluído tende a piorar o *fitness* para um valor mais próximo da média. No geral, nos dois cenários (estocástico ou não), a solução apresentada é mais robusta às falhas de componentes.

Como mencionado e analisado na Tabela 2, o tamanho diferente das soluções impacta de maneira diferente as falhas injetadas no circuito. Na análise de distribuição, podemos observar que para uma chance de falha de 1% no cenário binomial, a chance de falha em 1 porta no genoma evoluído era de 23.4%, enquanto que uma chance que se aproxima desse valor para a solução mínima é a chance de falha de 3%, onde a probabilidade é de 25.7%. Dessa forma, o mais correto ao se analisar a Tabela 3 é comparar o *fitness* do genoma evoluído com 1% de falha com a solução mínima com 3%, o que dá uma melhora de aproximadamente 10% na robustez.

6 Conclusão e Discussão

Este trabalho teve como objetivo principal explorar a aplicação da Programação Genética Cartesiana (CGP) na síntese de circuitos lógicos tolerantes a falhas, com foco em circuitos combinacionais básicos, como geradores de paridade ímpar. A evolução desses circuitos foi realizada em um cenário que simula falhas de componentes, permitindo a exploração de conceitos como redundância e degeneração, inspirados em sistemas biológicos, para aumentar a robustez dos circuitos.

Os resultados obtidos demonstraram que a estratégia evolutiva $(1+\lambda)$ foi eficaz na geração de circuitos robustos, capazes de manter sua funcionalidade mesmo na presença de falhas de componentes. A comparação entre o circuito evoluído e a solução mínima do estado da arte mostrou que o primeiro apresentou uma melhoria significativa na robustez, com um aumento de $18,9\%$ no *fitness* em cenários de falha única. No entanto, essa melhoria veio acompanhada de um aumento no tamanho do circuito, o que pode ser considerado uma desvantagem em termos de área e consumo de energia.

Em cenários estocásticos, onde a falha de componentes foi simulada com uma probabilidade variável, o circuito evoluído também se mostrou superior, embora a diferença em relação à solução mínima tenha sido menos expressiva. Isso sugere que, embora o circuito evoluído seja mais robusto, a relação entre o tamanho do circuito e a probabilidade de falha pode limitar sua eficácia em cenários onde múltiplas falhas ocorrem simultaneamente.

A estratégia de injeção de falhas durante o cálculo do *fitness* mostrou-se eficiente para guiar a evolução em direção a soluções mais tolerantes a defeitos. A utilização de um fator estocástico (ruído) também contribuiu para evitar a estagnação em máximos locais, permitindo que o algoritmo explorasse soluções inovadoras e menos óbvias.

Em termos de escalabilidade, o algoritmo evolutivo apresentou um desempenho computacional aceitável, com tempos de execução relativamente baixos para circuitos de pequena e média complexidade. No entanto, para circuitos mais complexos, o custo computacional pode se tornar um fator limitante, especialmente em cenários onde a avaliação do *fitness* envolve a simulação de múltiplas falhas.

Em conclusão, este trabalho demonstrou que a CGP é uma ferramenta promissora para a síntese de circuitos tolerantes a falhas, especialmente em contextos onde a robustez e a capacidade de adaptação são prioritárias. A abordagem proposta pode ser estendida para outros tipos de circuitos e aplicações, como somadores, multiplicadores e circuitos aritméticos mais complexos.

Por fim, os resultados obtidos reforçam a importância de se considerar a tolerância a falhas desde as etapas iniciais do projeto de circuitos, especialmente em um cenário onde a miniaturização e a complexidade dos sistemas digitais continuam a aumentar. A combinação de técnicas de síntese evolutiva com abordagens inspiradas em sistemas biológicos pode abrir novas perspectivas para o desenvolvimento de circuitos mais robustos e adaptativos, capazes de enfrentar os desafios impostos pela era da nanoeletrônica.

6.1 Trabalhos Futuros

Ao longo do desenvolvimento deste trabalho, algumas implementações e análises precisaram ficar de fora em detrimento do tempo de entrega. O principal ponto que certamente será trabalhado no futuro em próximas publicações é a análise do genoma evoluído na presença de redundância. A ideia fundamental é guiar a evolução para que o TMR seja explorado, garantindo que a saída do circuito final seja composta por 3 saídas idênticas (com a mesma função lógica), e com isso poderemos analisar mais a fundo como as técnicas de redundância, *interwoven* e degeneração podem impactar na robustez em comparação com a solução obtida através de CGP.

REFERÊNCIAS

AMARÚ, L. et al. New logic synthesis as nanotechnology enabler. *Proceedings of the IEEE*, IEEE, v. 103, n. 11, p. 2168–2195, 2015. <http://doi.org/10.1109/JPROC.2015.2460377>.

BURLYAEV, D. *Design, optimization, and formal verification of circuit fault-tolerance techniques*. Tese (Doutorado) — Université Grenoble Alpes, 2015. <https://tel.archives-ouvertes.fr/tel-01253368>.

CHEN, Y. et al. Sub-10 nm fabrication: methods and applications. *International Journal of Extreme Manufacturing*, IOP Publishing, v. 3, n. 3, p. 032002, 2021. <http://doi.org/10.1088/2631-7990/ac087c>.

DJUPDAL, A.; HADDOW, P. C. The route to a defect tolerant lut through artificial evolution. *Genetic Programming and Evolvable Machines*, Springer, v. 12, p. 281–303, 2011. <http://doi.org/10.1007/s10710-011-9129-2>.

EDELMAN, G. M.; GALLY, J. A. Degeneracy and complexity in biological systems. *Proceedings of the national academy of sciences*, National Acad Sciences, v. 98, n. 24, p. 13763–13768, 2001. <https://doi.org/10.1073/pnas.231499798>.

GREEN, C. et al. Trends in human spaceflight: Failure tolerance, high reliability and correlated failure history. maio 2019. Disponível em: <https://ntrs.nasa.gov/api/citations/20190025830/downloads/20190025830.pdf>.

MANAZIR, A.; RAZA, K. Recent developments in cartesian genetic programming and its variants. *ACM Computing Surveys (CSUR)*, ACM New York, NY, USA, v. 51, n. 6, p. 1–29, 2019. <https://doi.org/10.1145/3275518>.

MILANO, N.; NOLFI, S. Robustness to Faults Promotes Evolvability: Insights from Evolving Digital Circuits. *PLOS ONE*, v. 11, n. 7, p. e0158627 – 17, 2016. ISSN 1932-6203. <http://doi.org/10.1371/journal.pone.0158627>.

MILANO, N.; PAGLIUCA, P.; NOLFI, S. Robustness, evolvability and phenotypic complexity: insights from evolving digital circuits. *Evolutionary Intelligence*, v. 12, n. 1, p. 83–95, 2019. ISSN 1864-5909. <http://doi.org/10.48550/arXiv.1712.04254>.

PIERCE, W. H. Interwoven redundant logic. *Journal of the Franklin Institute*, Elsevier, v. 277, n. 1, p. 55–85, 1964. <https://doi.org/10.1016/0016-0032(64)90039-0>.

RYSER-WELCH, P.; MILLER, J. F. Ppsn 2016 tutorial: A graph-based gp and cartesian genetic programming. *University of York*, 2016. <http://doi.org/10.13140/RG.2.2.16335.69283>.

SOARES, Y.; TAVORA, H.; BLAWID, S. Optimizing cgp hyperparameters for efficient search of degenerate multiplexer designs. *Chip on the Cliffs, SForum Proc*, 2024. <https://sbmicro.org.br/sforum-eventos/sforum2024/OPTIMIZING%20CGP%20HYPERPARAMETERS%20FOR%20EFFICIENT%20SEARCH%20OF%20DEGENERATE%20MULTIPLEXER%20DESIGNS.pdf>.

UBAR, R. et al. Fault modelling with structural bdds. In: *Structural Decision Diagrams in Digital Test: Theory and Applications*. [S.l.]: Springer, 2024. p. 113–160. <http://doi.org/10.1007/978-3-031-44734-1_4>.