Emerson Carlos Pedrino

Sistema Automático para Geração de Circuitos Lógicos Utilizando Programação Genética Cartesiana

Trabalho de Conclusão de Curso apresentado à
Escola de Engenharia de São Carlos, da
Universidade de São Paulo

Curso de Engenharia Elétrica com ênfase em
Sistemas de Energia e Automação

ORIENTADOR: Prof. Dr. José Carlos de Melo Vieira Júnior

ALUNO: Emerson Carlos Pedrino

São Carlos
2015

Pedrino, Emerson Carlos
P371s Sistema automático para geração de circuitos lógicos utilizando programação genética cartesiana / Emerson Carlos Pedrino; orientador José Carlos de Melo Vieira Júnior. São Carlos, 2016.

Monografia (Graduação em Engenharia Elétrica com ênfase em Sistemas de Energia e Automação) -- Escola de Engenharia de São Carlos da Universidade de São Paulo, 2016.

1. Geração Automática de Circuitos Lógicos. 2. Programação Genética Cartesiana. 3. FPGA. I. Título.

FOLHA DE APROVAÇÃO

Nome: Emerson Carlos Pedrino

Título: "Sistema automático para geração de circuitos lógicos utilizando programação genética cartesiana"

Trabalho de Conclusão de Curso defendido e aprovado em 14 / 06 / 2016 ,

com NOTA 10,0 (dez , zero), **pela Comissão Julgadora:**

Prof. Dr. José Carlos de Melo Vieira Júnior - (Orientador - SEL/EESC/USP)

Prof. Associado Ivan Nunes da Silva - (SEL/EESC/USP)

Prof. Dr. Ricardo Quadros Machado - (SEL/EESC/USP)

Coordenador da CoC-Engenharia Elétrica - EESC/USP:
Prof. Dr. José Carlos de Melo Vieira Júnior

DEDICATÓRIA

Dedico este trabalho a meu pai Antonio (*in memorian*), a minha querida esposa Marly e a minha querida filha Letícia.

AGRADECIMENTOS

Agradeço a Deus, primeiramente, por todas as bênçãos concedidas em minha vida até o presente momento. Também, agradeço ao meu orientador Prof. Dr. José Carlos de Melo Vieira Júnior por sua bondade, amizade e ajuda durante este período extremamente feliz e de grandes realizações alcançadas. Por fim, agradeço a Igor Felipe Gallon, pela ajuda na implementação e na organização das idéias descritas neste texto, e aos meus familiares, em especial a minha esposa Marly, por sua compreensão durante este período de trabalho incessante.

EPÍGRAFE

"O primeiro lugar cabe aos que nascem com o saber. Vêm a seguir aqueles que têm de estudar para adquiri-lo. Depois, vêm os que só o adquirem à custa de grandes esforços. Quanto aos que não têm nem inteligência nem vontade de aprender, são esses os últimos homens".

Confúcio

SUMÁRIO

RESUMO

Algoritmos Evolutivos são algoritmos bastante eficientes quando usados para otimização de funções. São baseados nos conceitos da Biologia e da Teoria de Evolução de Darwin para solucionar problemas da área de Engenharia, entre outros. O objetivo deste trabalho de conclusão de curso foi construir um sistema inteligente capaz de gerar circuitos lógicos automaticamente, combinacionais e/ou sequenciais, que satisfaçam a imposições feitas pelo usuário, expressas por uma Tabela Verdade. Assim, são utilizadas técnicas de Programação Genética Cartesiana para a codificação e decodificação de circuitos digitais em forma de grafos. O algoritmo foi desenvolvido e testado no ambiente MATLAB, e para comparação dos resultados foi utilizada a *Toolbox* GPLAB, também para MATLAB. O melhor cromossomo obtido nesta etapa é convertido em *hardware*, por meio da linguagem Verilog-HDL, para implementação do resultado final em uma FPGA da família Cyclone II da Altera. Portanto, neste projeto foi desenvolvida uma arquitetura flexível em *hardware*, que é reconfigurada pelo melhor indivíduo gerado pelo algoritmo evolutivo desenvolvido no MATLAB. Além disso, blocos lógicos do tipo *Flip-Flop D* também foram implementados no sistema de *hardware* para geração de circuitos lógicos sequenciais. Por fim, foram realizados testes de execução do algoritmo para diferentes problemas envolvendo a geração automática de circuitos digitais e os resultados obtidos foram comparados a resultados similares encontrados na literatura.

Palavras-chave: Algoritmos Genéticos, Programação Genética Cartesiana, MATLAB, GPLAB, Circuitos Lógicos, FPGA.

ABSTRACT

Evolutionary algorithms are very efficient algorithms when used for function optimizations. They are based on the concepts of Biology and Darwin's Theory of Evolution to solve engineering problems, among others. The aim of this Undergraduate Final Project was build an intelligent system capable of generating logic circuits automatically, for satisfying impositions made by a user, expressed by a Truth Table. Thus, Cartesian Genetic Programming techniques are used for encoding and decoding digital circuits by means of graph theory. The algorithm was developed and tested in the MATLAB environment, and to compare the results it was used the Toolbox GPLAB, also for MATLAB. The best chromosome obtained in this step is converted to hardware, by means of Verilog-HDL language, for implementing the final result (best chromosome) into a FPGA from Altera. Therefore, in this project, a flexible architecture was developed in hardware, which is reconfigured by the best individual (best solution) generated by the evolutionary algorithm developed in MATLAB. Besides that, logic blocks containing D-type Flip-Flops were implemented in the hardware system for generating sequential logic circuits. Finally, they were carried out algorithm execution tests for various problems involving the automatic generation of digital circuits, and the results obtained were compared with similar results found in the literature.

Keywords: Genetic Algorithms, Cartesian Genetic Programming, MATLAB, GPLAB, Logic Circuits, FPGA.

1 Introdução

Neste trabalho de conclusão de curso (TCC) foram estudadas técnicas envolvendo Algoritmos Evolutivos (AEs) e Programação Genética Cartesiana (PGC), para a elaboração de um sistema automático, contendo módulos em *software* e em *hardware*, capaz de gerar circuitos lógicos (combinacionais e sequenciais) para determinadas aplicações. Considerando que a tarefa de projetar circuitos digitais não é trivial na prática para algumas aplicações complexas, muitas vezes até mesmo para um especialista, tal método permite gerar soluções de forma inteligente, facilitando-se assim o processo de projeto desses sistemas. Diferentemente de outros trabalhos encontrados na literatura, que se limitam apenas à geração automática de circuitos combinacionais em *software*, neste é proposta uma alternativa viável para implementação de módulos seqüências, além de uma arquitetura reconfigurável para implementação do sistema como um todo em *hardware* embarcado.

Os algoritmos utilizados neste processo foram implementados e testados em MATLAB (MATHWORKS, 1991), aproveitando-se dos recursos de cálculo vetorial e matricial oferecidos por essa ferramenta. Além disso, o algoritmo desenvolvido foi comparado a algumas funções da *Toolbox* GPLAB (SILVA, 2007), para MATLAB, na tentativa de melhorar o seu tempo de busca na resolução dos problemas apresentados.

Os códigos elaborados foram modularizados e divididos em arquivos do MATLAB, visando-se maior legibilidade, facilidade para detecção e correção de erros. Também, foram realizados estudos e testes de diferentes métodos de mutação e estratégias de evolução, a fim de se verificar quais foram os mais adequados para a situação em questão, melhorando-se assim, o desempenho de busca do algoritmo.

O resultado obtido nesta etapa foi então convertido em uma arquitetura de *hardware* flexível, por meio do uso da linguagem (VERILOG, 1996), e implementado em uma FPGA (*Field Programmable Gate Array*) da Altera (ALTERA, 2016).

As técnicas de evolução empregadas neste trabalho são encontradas na literatura de Algoritmos Genéticos, e a representação dos circuitos lógicos no algoritmo evolutivo é baseada nos conceitos de Programação Genética Cartesiana, descrita por Miller (2011).

A arquitetura flexível para FPGA foi desenvolvida em Verilog-HDL, e implementa, tanto circuitos puramente combinacionais, quanto sequenciais, que são fornecidos, de forma codificada, pelo Algoritmo Genético em MATLAB.

2 Algoritmos Evolutivos

Os Algoritmos Evolutivos (AEs) são modelos computacionais baseados na Seleção Natural e na Evolução das Espécies, idealizadas por Darwin (1859) no seu livro "A Origem das Espécies".

Os AEs são úteis quando o problema proposto é inviável em relação ao tempo de processamento que o computador leva para resolver o problema, se utilizados métodos determinísticos convencionais, como, por exemplo, a força bruta (HOLLAND, 1992).

No problema do Caixeiro Viajante, onde um Caixeiro precisa percorrer um número definido de cidades para entregar suas mercadorias e gastar o mínimo com a viagem (GOMES, 2015), é necessário encontrar a menor rota possível que passe por todas as cidades. Se utilizado um método de força bruta para tentar encontrar a solução desse problema, devem-se testar todas as possibilidades de rota e para cada uma delas calcular a distância percorrida. Como o número de rotas possíveis é calculado pela quantidade de permutações do número de cidades, a quantidade de caminhos possíveis cresce exponencialmente ($n!$ sendo n o número de cidades) e o tempo necessário para um computador realizar o cálculo torna o problema impossível de ser solucionado quando *n* é muito grande.

Os AEs são mais viáveis quando o espaço de busca é muito grande, conforme visto, e utilizam propriedades estatísticas para chegarem mais perto de uma possível solução do problema (DAVIS, 1991).

Por sua vez, os Algoritmos Genéticos (AGs) são Algoritmos Evolutivos que utilizam sobre uma determinada população de indivíduos operadores genéticos como cruzamento, mutação e seleção, a fim de obterem indivíduos mais aptos ao passar do tempo. Define-se um critério de avaliação e a cada geração esses indivíduos passam por esse processo recebendo uma "nota", cujos melhores indivíduos (possíveis soluções do problema) são selecionados por alguma estratégia de evolução depois de aplicados os operadores genéticos (mutação e cruzamento). Quem sobrevive após a seleção tem maiores chances de dar continuidade as suas características e gerarem descendentes parecidos. Desta forma, a população se torna cada vez mais adaptada ao problema e a probabilidade de se encontrar um indivíduo com as propriedades que satisfaçam o problema proposto inicialmente se torna maior.

A estrutura básica de um Algoritmo Genético pode ser descrita da seguinte forma (MITCHELL, 1999):

1. Criação aleatória de uma população inicial;
2. Cálculo da aptidão de cada indivíduo;
3. Escolha de alguns indivíduos para se reproduzirem (os pais);
4. Operações de cruzamento entre os pais e mutação na população para geração de novos descendentes;

5. Caso o tempo desejado não seja obtido ou o indivíduo que solucione o problema não seja encontrado, voltar ao passo 2.

Como esses algoritmos utilizam fatores aleatórios para operarem, pode-se encontrar uma solução diferente cada vez que eles são executados. Com isso, eles se diferem dos algoritmos exatos e não garantem encontrar a solução mais otimizada de todo o espaço de busca (máximo global), mas chegam bem perto dela, haja vista que a cada geração, busca-se melhorar a aptidão dos indivíduos (LINDEN, 2012).

2.1 Codificação dos indivíduos

Todos os indivíduos na natureza apresentam no interior de suas células o DNA (em inglês: *deoxyribonucleic acid*; ou ADN em português: *ácido desoxirribonucleico*), responsável por carregar informações que caracterizam o funcionamento do organismo utilizando um sistema de codificação composto por quatro bases nitrogenadas (adenina, citosina, guanina e timina).

Denomina-se **cromossomo** uma longa seqüência de fita de DNA e **genes** os pedaços de cromossomo responsáveis por características específicas de um indivíduo, como, por exemplo, a cor dos olhos, do cabelo, etc. Um conjunto específico de genes é chamado de **genótipo** e dá origem ao **fenótipo**, que são as características físicas dos indivíduos dadas pela informação genética contida no interior dos genes.

Assim como na Biologia, os indivíduos com os quais os AGs trabalham também são representados por cromossomos. Logo, em Engenharia, os cromossomos são vetores de números - sejam eles binários, inteiros ou reais - que decodificam-se em um fenótipo. Avaliando-se esse fenótipo, pode-se então calcular a aptidão daquele indivíduo para o problema no qual o algoritmo está buscando uma solução. A Figura 1 apresenta um cromossomo codificado em números binários, no qual cada gene possui 4 *bits*.

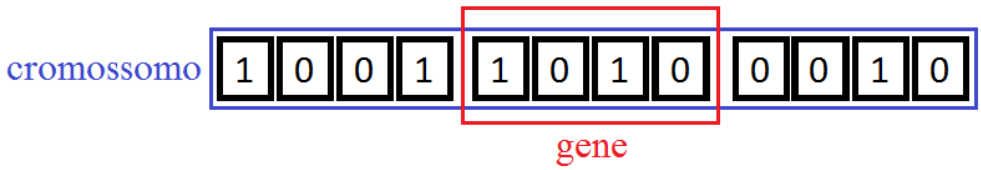

Figura 1: Representação de um indivíduo do Algoritmo Genético utilizando o sistema binário.

Para cada tipo de problema, é necessário um tipo adequado de representação e como ele será codificado na forma de cromossomos. Procura-se, então, escolher o que for mais fácil de implementar e o mais próximo da realidade possível.

2.2 Função de Avaliação

A Função de Avaliação, também chamada de Função de Aptidão, é responsável por avaliar um indivíduo dentro de uma população e informar o quão próximo da solução desejada ele se encontra.

O cálculo deve abordar ao máximo as características de um indivíduo para que a nota atribuída a ele o represente bem no contexto do problema a ser solucionado. Alguns fatores devem ser analisados com cuidado quanto à escolha de como será a função de avaliação, pois refletem diretamente no avanço e na taxa de variabilidade genética durante o processo de busca do AG, entre eles, o chamado Super Indivíduo (LINDEN, 2012).

O Super Indivíduo é aquele que depois de calculada a aptidão de todos os indivíduos, sua avaliação é superior em relação aos demais indivíduos, fazendo com que este seja sempre promovido a pai. Dessa forma, o número de filhos com carga genética deste indivíduo aumenta ao passar das gerações e a variabilidade genética diminui, fazendo com que a população convirja de forma prematura a um ponto do espaço de soluções (máximo ou mínimo local).

Por outro lado, se o índice de variabilidade genética é muito alto, os indivíduos gerados na próxima geração se tornam muito divergentes em relação à geração anterior e o algoritmo passa a funcionar semelhantemente a uma busca por força bruta (DAVIS, 1991).

2.3 Métodos de Cruzamento

O operador de cruzamento, responsável por transmitir as características de pais para filhos, é essencial para que o conceito de evolução seja aplicado ao AG.

Se dois indivíduos possuem boas aptidões para solucionar um problema, um indivíduo gerado a partir da mistura dessas qualidades tem chances também de ter uma boa aptidão nesse contexto (GOLDBERG, 1989). Isso faz com que a geração sucessora seja, de modo geral, melhor que a antecessora, e assim com o passar do tempo a população convirja para uma solução satisfatória (HOLLAND, 1992).

O **cruzamento de um ponto** é a maneira mais simples de realizar tal processo. A técnica consiste em escolher arbitrariamente uma posição do cromossomo, denominado ponto de corte, e separar os pais em duas partes cada, com isso, realiza-se a permutação dessas partes gerando novos filhos conforme a Figura 2.

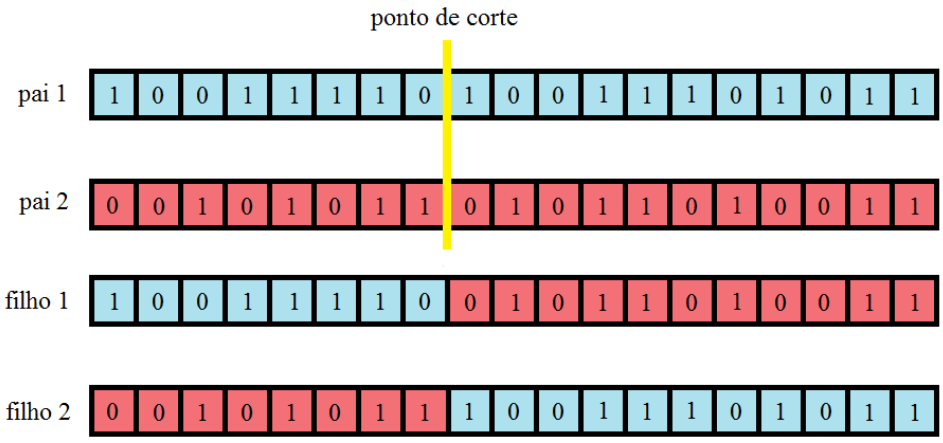

Figura 2: Demonstração do processo de cruzamento de um ponto.

Também existe o método de **cruzamento de dois pontos**, além de outras opções existentes (SIVANANDAM; DEEPA, 2007).

2.4 Mutação

O operador de mutação altera, baseado em uma dada probabilidade, genes em posições aleatórias do cromossomo, com a finalidade de se manter a diversidade no processo de evolução (LINDEN, 2012).

Neste processo, deve-se tomar o cuidado para que o valor a ser reposto no gene esteja dentro dos domínios de solução do problema sob consideração, para que o indivíduo após a mutação ainda seja válido.

Na Figura 3 é demonstrado o processo de mutação realizado em um cromossomo de números inteiros, utilizando-se uma taxa de mutação de 20% (0,2). Os valores utilizados para compor o indivíduo estão entre 1 e 20. Por exemplo, no décimo gene (19) o valor aleatório gerado (0,13) foi menor que a taxa de mutação, então, este foi alterado para (5) após sua mutação.

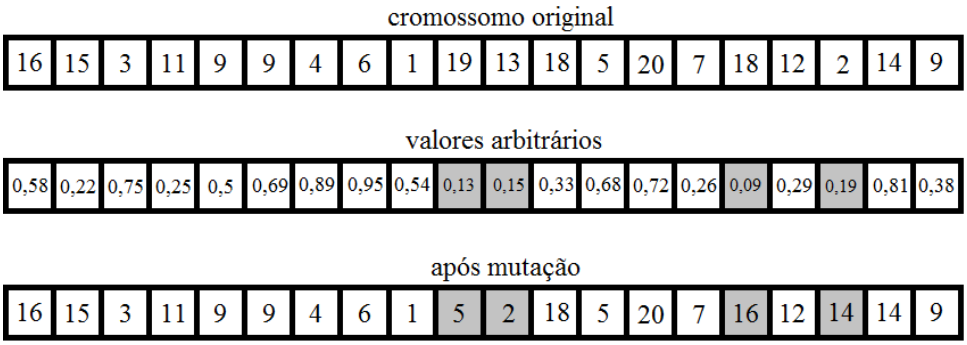

Figura 3: Operação de mutação em cromossomo de codificação inteira com valores compreendidos entre 1 e 20. A taxa de mutação adotada neste exemplo foi de 20%.

Uma taxa de mutação muito elevada pode fazer com que o algoritmo demore para encontrar uma solução satisfatória. Se, por exemplo, ela for de 100%, é como se um novo indivíduo fosse criado com características aleatórias, já que todos os seus genes sofreriam mutação. Os indivíduos modificados não carregariam consigo informações genéticas das gerações passadas e a busca não teria uma direção definida a ser seguida.

2.5 Estratégias de Evolução

2.5.1 Elitismo

A estratégia de evolução denominada elitismo diz que os n melhores indivíduos de uma geração devem sempre sobreviver para a próxima geração (DAVIS, 1991), possibilitando que se tenha um crescimento na avaliação da população com o passar do tempo, visto que a avaliação do melhor indivíduo daquela população será melhor que a avaliação do melhor indivíduo da geração antecessora ou no mínimo igual, se considerar o pior caso em que nenhum melhor indivíduo é criado.

2.5.2 Estratégia $\mu + \lambda$

A estratégia $\mu + \lambda$ diz que para cada geração, escolhem-se μ indivíduos denominados pais para gerarem λ filhos, sendo $\mu < \lambda$. A população formada por $\mu + \lambda$ indivíduos competem entre si e novamente são escolhidos μ indivíduos, geralmente os mais adaptados, para sobreviverem até a próxima geração (LINDEN, 2012).

O que pode ocorrer nesse tipo de estratégia é a possibilidade da convergência prematura da população para um mínimo ou máximo local, visto que se os λ filhos gerados tiverem poucas mudanças em relação aos pais (porque os filhos são frutos de operação de cruzamento, onde partes genéticas dos pais

são copiadas), suas avaliações serão muito próximas, reduzindo-se a variabilidade genética de uma geração para outra e tendendo a estagnar os indivíduos em um ponto do espaço de soluções.

2.5.3 Steady State

Como a população utilizada pelo Algoritmo Genético é controlada, e na maioria dos casos fixa, toda vez que ocorre a evolução para a próxima geração, a geração anterior inteira morre para dar espaço aos novos indivíduos. Com isso, a estratégia *Steady State* gera e substitui aos poucos os indivíduos de uma população, permitindo que cromossomos de gerações diferentes possam ainda se cruzar (MITCHELL, 1999; LINDEN, 2012).

Por exemplo, substituir apenas alguns piores indivíduos da geração corrente por filhos resultantes de cruzamentos entre os melhores pais da geração anterior. Ou ainda, selecionar aleatoriamente alguns cromossomos para serem substituídos por outros, em vez de escolher somente os piores. Manter alguns indivíduos de menores avaliações auxilia na preservação da variabilidade genética.

Há ainda a possibilidade de cruzamento entre pais e filhos da mesma geração, porém isso traz um risco ao Algoritmo Genético de haver uma convergência genética antecipada, visto que a recombinação de genes entre pai e o próprio filho não resulta em diferenças significativas.

3 Programação Genética

A Programação Genética é uma técnica que tem se mostrado promissora atualmente, porque diferentemente dos Algoritmos Genéticos que produzem soluções fixas, essa técnica gera programas (soluções mais flexíveis) que solucionam diversos problemas práticos de diversas áreas do conhecimento (KOZA, 1992). A representação dos indivíduos na Programação Genética é feita por estruturas conhecidas como **árvores**.

Os **nós** das árvores são campos de memória que podem armazenar valores usados como entrada do programa ou também funções. Cada nó da árvore representa um gene do cromossomo (CELES; CERQUEIRA; RANGEL, 2004).

A **Função de Aptidão** na Programação Genética calcula a diferença entre a saída obtida por aquele cromossomo e a saída desejada. Para retornar um valor numérico, a Função de Avaliação precisa entrar com o valor, executar o programa e verificar sua saída (KOZA, 1992). Entende-se executar o programa como sendo a tarefa de decodificar o cromossomo.

A **mutação** não altera apenas um nó da árvore, mas sim uma sub-árvore toda (MICHALEWICZ, 1992; MITCHELL, 1999; POLI et al., 2008) (Figura 4). Dessa forma, o operador genético gera estruturas válidas para serem repostas onde se deseja realizar a mutação do gene. Isso faz com que se altere a estrutura da árvore, porém garante que os indivíduos ainda continuem dentro dos parâmetros permitidos.

O operador genético de **cruzamento** utiliza a mesma ideia de substituição de sub-árvores: seleciona aleatoriamente uma sub-árvore do primeiro e do segundo pai, e comuta entre eles para gerar novos filhos (EIBEN; SMITH, 2003).

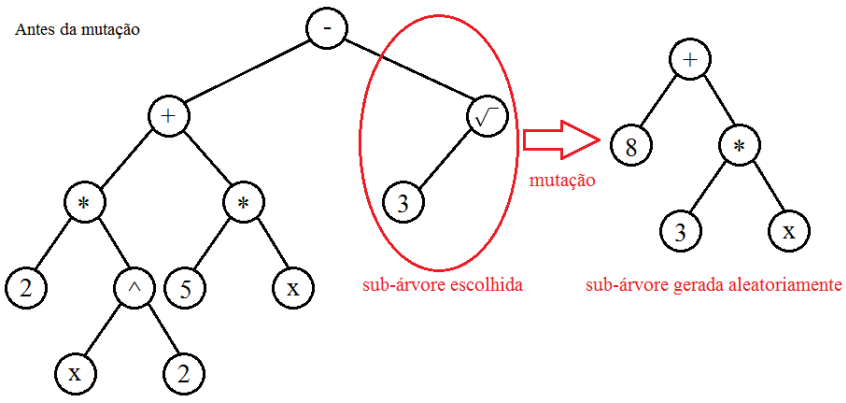

Figura 4: Processo de mutação. Sub-árvore escolhida aleatoriamente para sofrer mutação. O operador cria uma nova sub-árvore válida para substituir.

A Programação Genética funciona da mesma forma que os Algoritmos Evolucionários, sendo apenas necessário modificar algumas operações realizadas, como a Função de Avaliação, mutação e cruzamento, para manipular a estrutura de dados que a Programação Genética usa para representar seus indivíduos.

3.1 Programação Genética Cartesiana

A Programação Genética Cartesiana (PGC) é uma derivação da Programação Genética porque também trabalha com a geração de programas solucionadores de problemas, porém, a representação utilizada para manipular os indivíduos não são estruturas de árvores. Os indivíduos nesse caso são representados por grafos (MILLER, 2011).

Os grafos são estruturas de dados muito utilizadas em computação e possuem dois conjuntos: um conjunto de vértices ou nós, e um conjunto de arestas (RUOHONEN, 2010).

A representação dos indivíduos utilizando grafos na Programação Genética Cartesiana acontece da seguinte maneira: os grafos utilizados são indexados, ou seja, cada nó possui um índice. Os nós representam os genes dos cromossomos. Cada nó guarda o índice de uma função que segue uma tabela de funções predefinida e os índices dos outros genes que se interligam a ele. Organizam-se em linhas e colunas, e os nós se conectam apenas entre colunas, como se cada uma fosse um nível do grafo. Os nós da primeira coluna se conectam com as entradas do programa e as saídas dos genes da última coluna se conectam com as saídas do sistema. O número máximo de níveis que cada gene pode se conectar é chamado de "*level-back*" e é representado pelo parâmetro l. Se $l = 2$, por exemplo, os genes podem se conectar com no máximo duas colunas anteriores.

O número de linhas n_r, o número de colunas n_c, o número de entradas n e o número de saídas m são parâmetros definidos antes de se iniciar a evolução, sendo que n_r e n_c são parâmetros que determinam quantos genes cada indivíduo possuirá e consequentemente o tamanho de cada cromossomo, onde o número máximo de nós alocados por indivíduo será: $L_n = n_r \times n_c$. Além disso, é necessário conhecer o número de funções n_f que serão utilizadas na codificação genética.

Na Figura 5, encontra-se o esquema geral proposto por Miller (2011) de um mapeamento genético, a organização dos grafos e as regras para se determinar os índices na codificação.

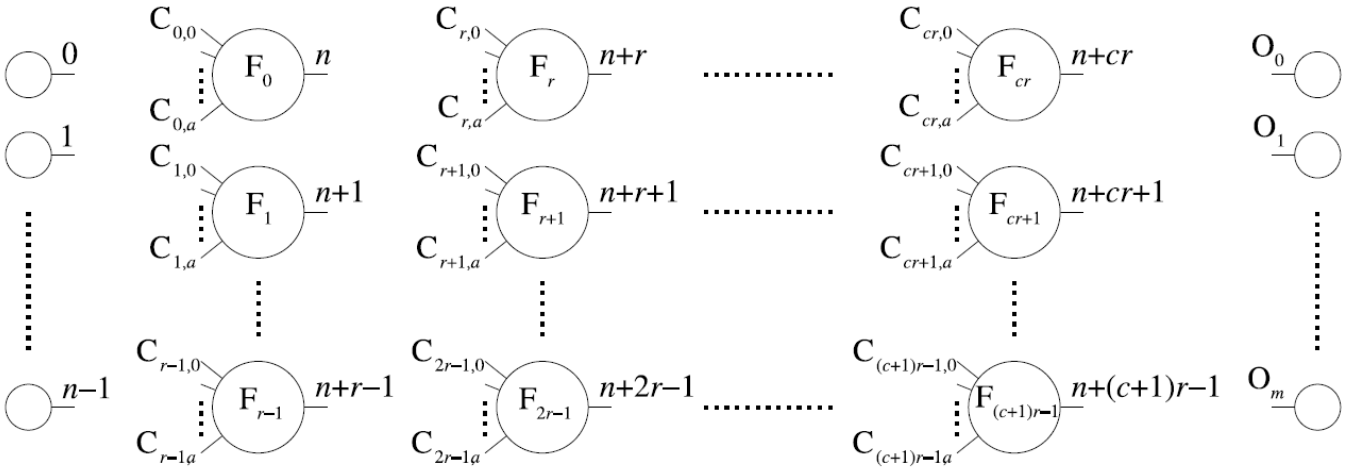

Figura 5: Resumo do esquema de organização dos grafos e índices (MILLER, 2011).

Os nós que vão do índice *0* até $n-1$ são as entradas do programa e os nós $O_0, O_1, \ldots, O_m$ são as suas saídas. Os demais são nós que representam os genes do cromossomo, com suas entradas *C* e saídas que vão dos índices n até $\mathrm{n} + (\mathrm{c}+1)\mathrm{r} - 1$, de acordo com a figura.

Na figura, C_{ia} representa as entradas de cada gene, onde i representa o índice do nó referido e a indica qual entrada do gene é referenciada, também chamado de *aridade*. Por exemplo, C_{31} representa a segunda entrada (pois a contagem se inicia do *0*) do gene de índice 3. O termo r é uma constante que representa o número de linhas que a rede tem e c representa o índice da coluna a qual o gene está localizado, começando ambos em *0*.

O vetor de índices que representa a rede acima é construído então com sucessivas *n-uplas*, onde cada uma representa um nó da rede e contém sempre o endereço da função utilizada na tabela de funções; os endereços dos nós que servem de entrada para a função e as saídas do programa podem ser vistos na Figura 6.

$$\mathrm{F_0C_{0,0}....C_{0,a}F_1C_{1,0}....C_{1,a}}\ldots\ldots\ldots\ldots\mathrm{F_{(c+1)r-1}C_{(c+1)r-1,0}....C_{(c+1)r-1a}O_0O_1....O_m}$$

Figura 6: Vetor de índices que codifica a rede de nós e suas conexões (MILLER, 2011).

Para a construção da rede, os índices precisam seguir algumas regras e intervalos de valores permitidos conforme o nível do grafo. Como a criação da população inicial é feita de forma aleatória, os índices gerados devem ser concisos para manterem a organização da estrutura de nós.

Sendo f_n o número de funções existentes na tabela de funções, o índice f_i usado em cada nó i deve estar dentro do intervalo:

$$0 \leq f_i \leq f_n \tag{1}$$

Considerando-se que os nós estão na coluna j e que C_{ik} são as entradas k desses nós, tem-se o seguinte intervalo:

$$Se\ l \leq j, \qquad n + (j - l)n_r \leq C_{ik} \leq n + jn_r \tag{2}$$

$$Se\ l > j, \qquad 0 \leq C_{ik} \leq n + jn_r \tag{3}$$

Os índices O_s das saídas do programa que se conectam com as saídas dos nós seguem a seguinte restrição:

$$0 \leq O_s \leq n + L_n \tag{4}$$

A Figura 7 exemplifica a utilização dessa organização de nós obedecendo às restrições impostas para a geração de funções matemáticas e a codificação correspondente no vetor de endereços, considerando $n = 2,\ m = 2, n_r = 2, n_c = 2, l = 2$ e além disso, considerando a Tabela 1 de funções aritméticas.

Tabela 1: Funções aritméticas usadas no exemplo.

Índice	**Operação**
0	Adição
1	Subtração
2	Multiplicação
3	Divisão

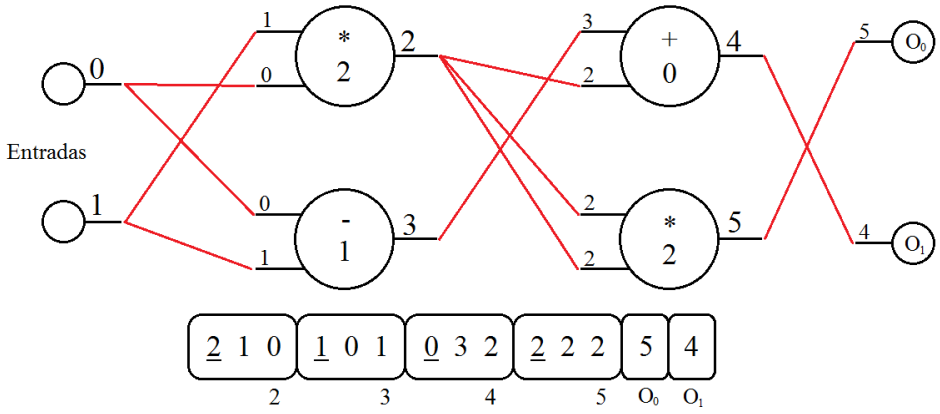

Figura 7: Exemplo de aplicação utilizando grafos indexados e sua respectiva codificação no cromossomo.

3.2 Outros modelos de PGC na literatura

Há ainda outros modelos de Programação Genética Cartesiana presentes na literatura que não foram utilizados neste trabalho. Dentre deles, a Programação Genética Modular e a Programação Genética Cartesiana Auto-Modificada (tradução para *Self-Modifying Cartesian Genetic Programming*).

A **Programação Genética Cartesiana Modular** (PGCM) trata do modelo de PGC clássica utilizando uma nova forma de representação dos genes dos indivíduos, a representação modular (WALKER; MILLER, 2008).

Os módulos encapsulam porções de genes do cromossomo representado pela PGC convencional. Esses módulos podem ser reaproveitados dentro do cromossomo, ou seja, segmentos de genes podem ser replicados e os nós são classificados em: funções primitivas (tipo 0), módulos contendo uma parte original do cromossomo (tipo 1) e módulos já reutilizados contendo uma parte replicada do cromossomo (tipo 2).

A **Programação Genética Cartesiana Auto-Modificada** (PGCAM) pode ser comparada a uma PGC na qual a rede de nós possui apenas uma linha (HARDING; MILLER; BAHNZAF, 2009).

Cada nó da rede de grafos possui além dos três parâmetros convencionais da PGC, outros três valores P_0, P_1e P_2 utilizados na codificação da função utilizada por cada nó e também um outro valor que indica se o nó pode ou não ser utilizado como saída do cromossomo.

Os índices de entradas dos nós utilizados são endereçados de forma relativa e indicam quantos nós anteriores há conectados ao nó atual. Por exemplo, um nó com entrada de índice -2 indica que ele se conecta com o segundo nó imediatamente anterior a ele.

O principal destaque da PGCAM é que as funções utilizadas na codificação têm também o poder de modificar a própria estrutura do genótipo e não só alterar seus valores numéricos. Os parâmetros P_0, P_1 e P_2 são utilizados nas funções que alteram o funcionamento do próprio programa e consequentemente o genótipo. Na Figura 8 é mostrado um exemplo de cromossomo gerado utilizando a PGCAM, onde os nós INP representam as entradas do programa e o nó $DADD$ representa, por convenção, a saída. O operador DUP (Figura 8a) insere na próxima iteração (Figura 8b) uma duplicação de um trecho do programa.

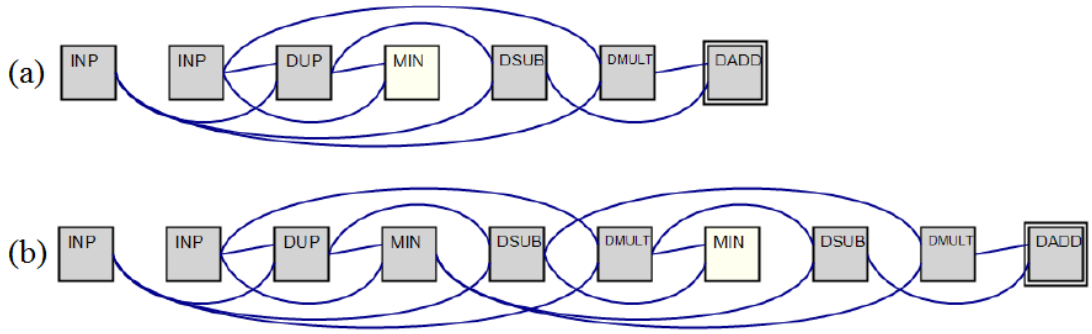

Figura 8: Exemplo de cromossomo gerado utilizando a representação em PGCAM. O operador DUP (a) duplica uma seção do cromossomo na próxima iteração (b) (HARDING; MILLER; BAHNZAF, 2009).

4 Algoritmo em MATLAB para a construção de Circuitos Lógicos

O algoritmo para geração automática de circuitos lógicos, proposto neste TCC, foi desenvolvido em linguagem de programação para MATLAB. Porém, esta linguagem não permite a utilização de índices *0*, então, todo o esquema de organização e limites de índices permitidos apresentados por Miller (MILLER, 2011) foram refeitos para contornar esse problema.

O algoritmo evolutivo proposto utiliza quatro primitivas lógicas e uma função neutra: AND, OR, XOR, NOT e NOP. A função NOP não realiza nenhuma operação lógica, apenas repassa o valor da primeira entrada de um gene para a saída correspondente.

Todas as funções descritas nas seções seguintes foram implementadas em MATLAB, conforme já supracitado. A seguir, encontra-se a tabela de funções (Tabela 2) com seus respectivos índices utilizados na codificação dos cromossomos:

Tabela 2: Tabela de funções utilizadas para a criação dos circuitos.

Índice	Função
1	AND
2	OR
3	XOR
4	NOT
5	NOP

A estrutura do programa desenvolvido, com suas interfaces relacionadas, pode ser vista na Figura 9. O usuário informa os parâmetros necessários, via interface de texto ou gráfica, para o funcionamento do AE, como quantidade de indivíduos da população, número máximo de gerações, taxa de mutação, tamanho da rede de nós e a Tabela Verdade com as entradas e saídas desejadas; também, informa o tipo de circuito a ser gerado (combinacional ou sequencial) e a quantidade de estados da máquina, caso o circuito a ser gerado seja sequencial.

Depois de encontrada a solução, o AE informa o cromossomo pai à interface que se encarrega de gerar o fenótipo na tela.

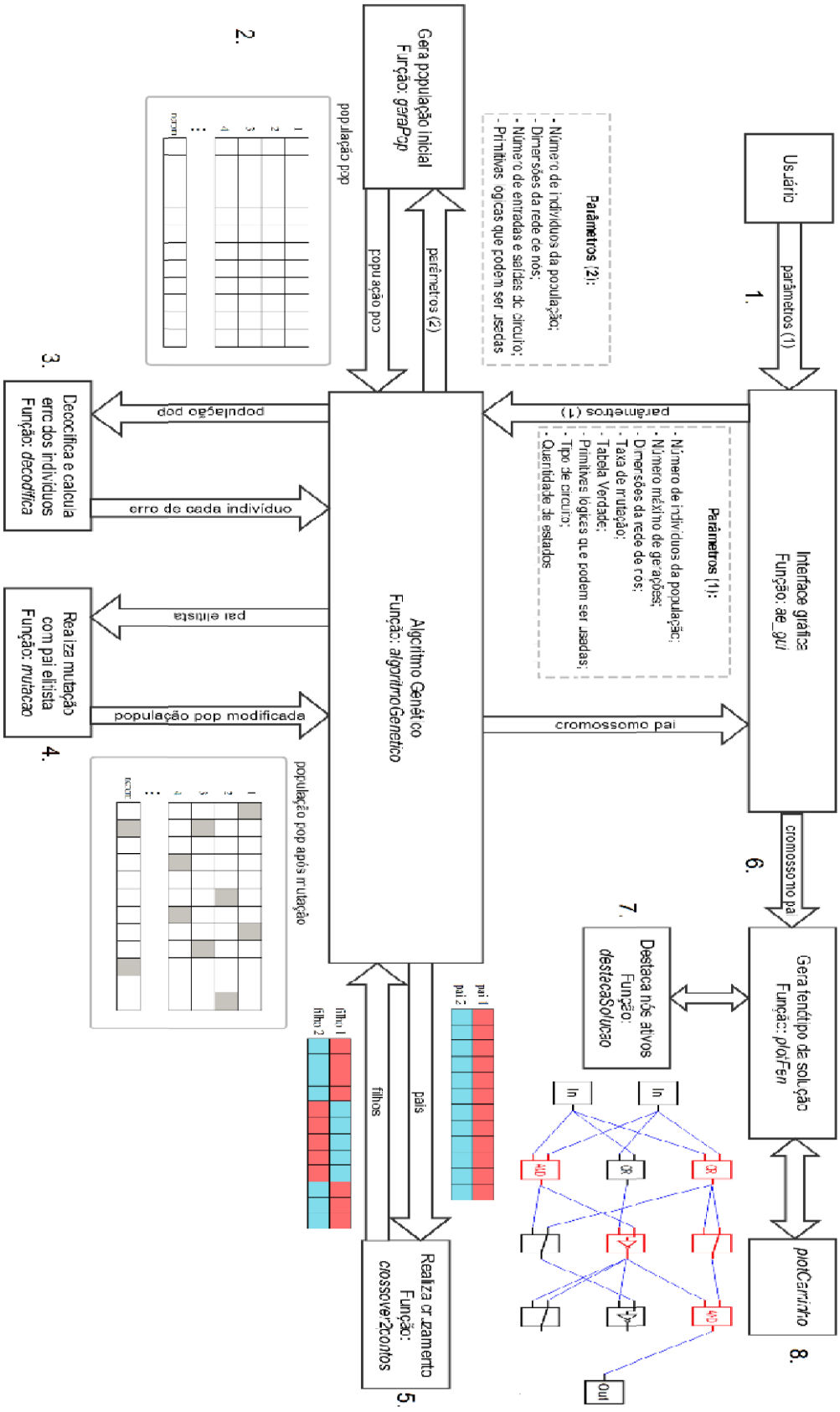

Figura 9: Estrutura geral do Algoritmo Evolutivo proposto para Construção Automática de Circuitos Digitais.

4.1 População inicial – Função $geraPop$

Para contornar o problema com índices nulos do MATLAB, foram refeitas as restrições (1), (2), (3) e (4), apresentadas na seção 4. Além disso, definiu-se que os nós de uma coluna só poderiam se conectar com os nós da coluna imediatamente anterior, ou seja, $l = 1$. Dessa forma, as novas restrições são:

$$1 \leq f_i \leq f_n \tag{5}$$

$$Se\ j = 0, \qquad 1 \leq C_{ik} \leq n \tag{6}$$

$$Se\ j > 0, \qquad (j-1)n_r + 1 \leq C_{ik} \leq jn_r \tag{7}$$

$$(n_c - 1)n_r + 1 \leq O_s \leq L_n \tag{8}$$

Onde: f_n é o número de funções lógicas; C_{ik} é o valor de entrada de cada gene i; k é limitado em 1 ou 2 (entradas 1 ou 2, respectivamente); n é o número de entradas do circuito lógico; n_r e n_c são o número de linhas e colunas, respectivamente, da rede de nós; O_s representa os valores de saída s do circuito e $L_n = n_r \times n_c$. Observa-se que a primeira coluna ($j = 0$) é tratada diferentemente das demais, pois os nós dessa só se conectam com as entradas do programa. Além disso, a enumeração dos genes começa no primeiro nó com valor 1, diferentemente da organização proposta por Miller.

A seguir (Figura 10), encontra-se o processo de criação da população inicial realizado pela função $geraPop$ (Figura 9). Para cada gene, de cada indivíduo da população, gera-se um valor aleatório f_i para as funções, e verifica-se se a coluna atual é a coluna 0; se for, utilizam-se as entradas do circuito lógico para preencher os valores de entrada da função (e_{i1} e e_{i2}), senão, utilizam-se as saídas dos genes da coluna anterior. Após todos os genes serem devidamente preenchidos com valores do domínio de soluções, geram-se valores para as saídas O_m do circuito, conforme a restrição (8).

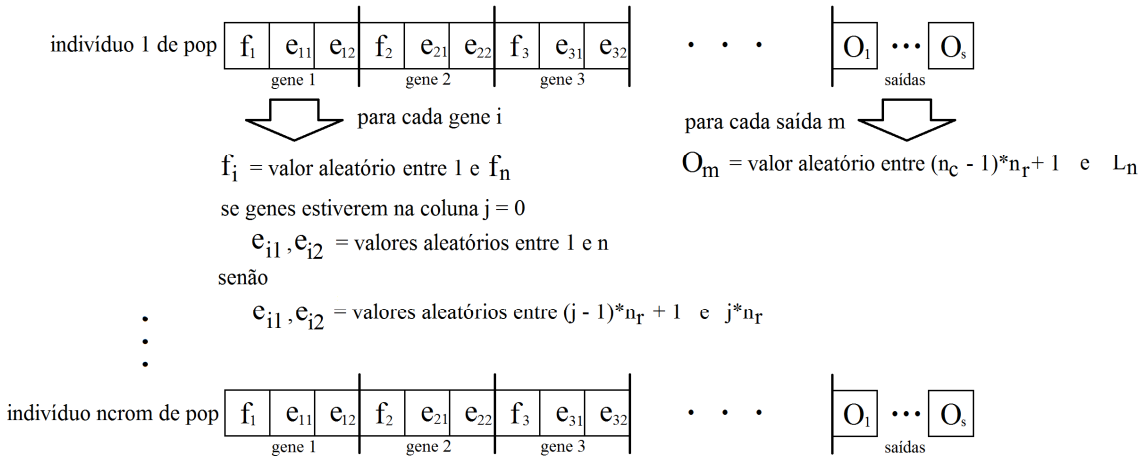

Figura 10: Diagrama funcional para a geração da população inicial com a função $geraPop$.

Após a execução da função $geraPop$, obtém-se a matriz pop (Figura 9) preenchida com os valores permitidos conforme os parâmetros de entrada, onde cada linha desta representa a codificação de um indivíduo. Para uma rede de 2 linhas e 3 colunas, para gerar um circuito lógico de 2 entradas e 1 saída, utilizando-se todas as primitivas lógicas disponíveis, a função *geraPop* pode gerar, por exemplo, um indivíduo representado pelo vetor de 19 posições ($3 \times n_r \times n_c + s = 3 \times 2 \times 3 + 1 = 19$), dado de acordo com a Tabela 3 a seguir.

Tabela 3: Cromossomo em forma de vetor de índices.

4	2	1	**3**	1	1	**5**	2	2	**4**	1	1	**4**	4	3	**1**	3	3	**6**

Gene 1	Gene 2	Gene 3	Gene 4	Gene 5	Gene 6	
Coluna j = 0		Coluna j = 1		Coluna j = 2		Saída
Linha 1	Linha 2	Linha 1	Linha 2	Linha 1	Linha 2	

O último elemento da tabela indica qual gene está conectado à saída do circuito, sendo sua representação em forma de grafo dada pela Figura 11, neste caso hipotético.

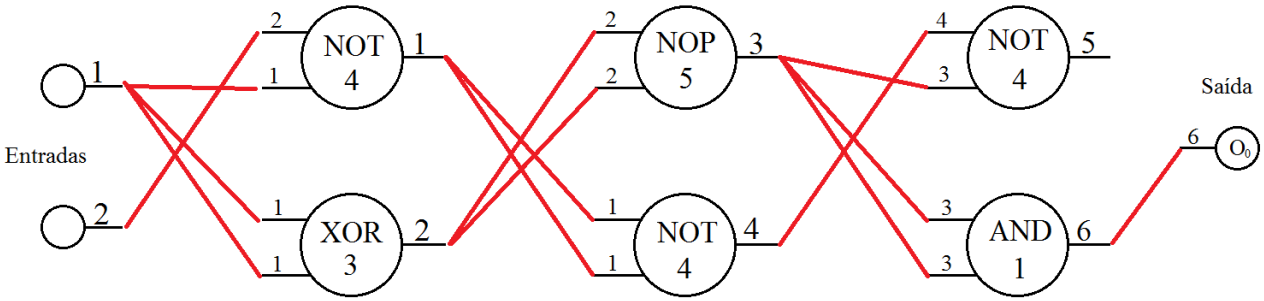

Figura 11: Representação do cromossomo gerado em forma de grafos indexados conforme limites adotados em (5), (6), (7) e (8).

4.2 Decodificação e Função de Aptidão – Função *decodifica*

O próximo passo, então, consiste em realizar a decodificação e o cálculo de custo para cada indivíduo da população corrente. Neste processo, os valores de entrada são recebidos, o circuito codificado em cada cromossomo é resolvido e suas saídas são geradas.

Os valores de entrada vêm da Tabela Verdade que representa o circuito em questão, que deve ser fornecida pelo usuário. O número de linhas k que corresponde às combinações possíveis de valores de

entrada é calculado por 2^n, sendo n o número de entradas do circuito. Para cada combinação de valores de entrada, o circuito gera os valores respectivos de saída.

Inicialmente, verificam-se se os genes são da coluna inicial, pois, os valores de entrada utilizados (e_{i1} e e_{i2}) são diretamente atribuídos pelas entradas da Tabela Verdade. Caso contrário, as entradas da função recebem os valores de saída dos nós da coluna imediatamente anterior. Após definidos os valores de entrada, aplica-se a função adequada f_i para cada gene e armazena-se cada resultado em uma matriz de saídas de genes (Figura 12).

Com isso, calcula-se o somatório do módulo da diferença entre os valores obtidos do circuito gerado e os valores desejados, dividindo-se pela quantidade de valores de saída que a Tabela Verdade possui, e multiplica-se o valor encontrado por 100. Assim, a função $decodifica$ (Figura 9) retorna para cada indivíduo uma porcentagem de erro. Por exemplo, um indivíduo que possui 2 entradas e 2 saídas, com erro de 12,5%, significa que há apenas 1 *bit* na saída diferente da solução proposta, já que existem 8 valores de saída (2^2combinações de entradas $\times$ 2 saídas) e $\frac{1}{8} = 0{,}125 = 12{,}5\%$.

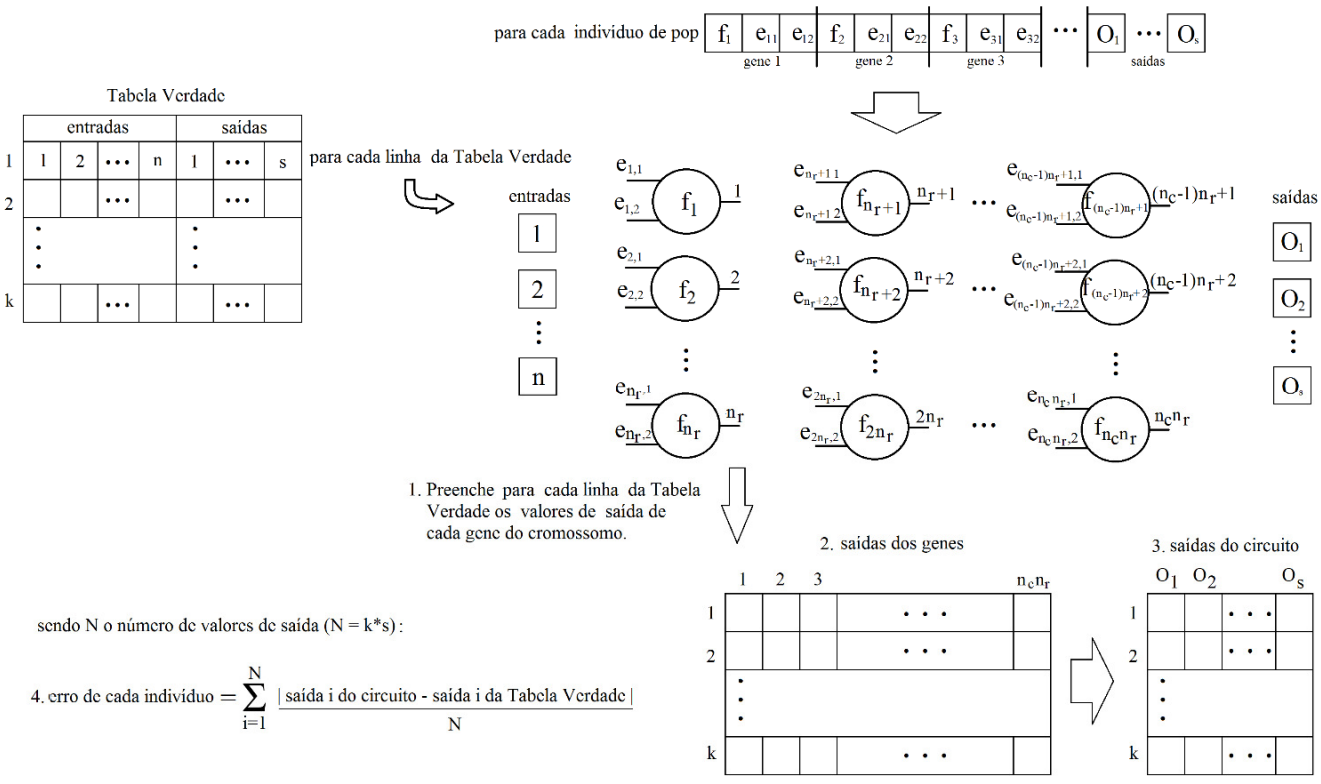

Figura 12: Diagrama de funcionamento da função $decodifica$.

4.3 Mutação – Função *mutacao*

O processo de mutação utilizado neste TCC funciona de forma semelhante ao processo descrito na Seção 2.4. Na Figura 13 é apresentado um exemplo. Inicialmente, a função *mutacao* (Figura 9) gera um valor pseudo-aleatório entre 0 e 1 para cada posição do cromossomo de cada indivíduo. Se ele for menor ou igual à Taxa de Mutação, aquela posição do vetor deverá sofrer mutação. Assim, gera-se um novo valor que obedeça aos limites (5), (6), (7) e (8). Caso contrário, aquela posição do vetor recebe a mesma posição, mas do cromossomo pai.

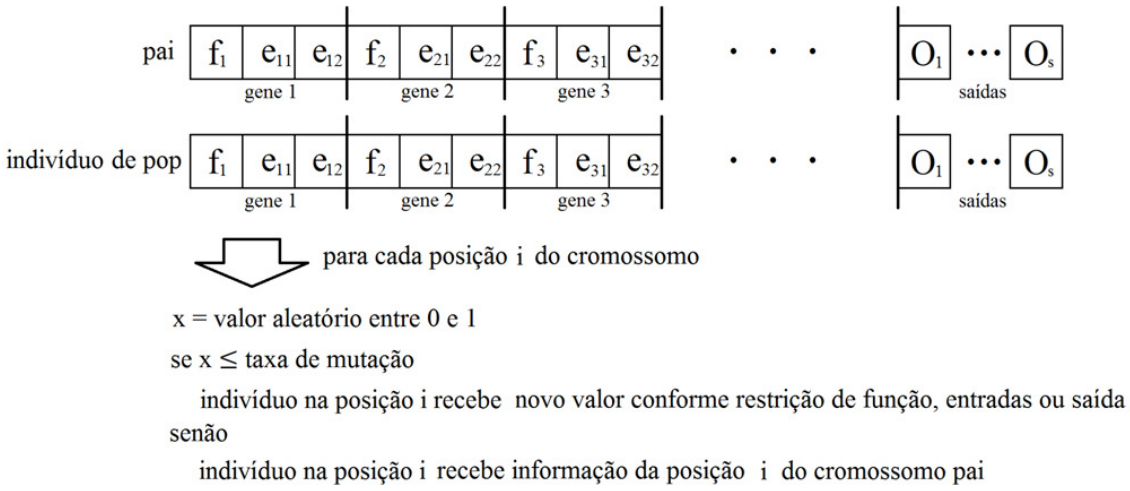

Figura 13: Diagrama de funcionamento do processo de mutação dos cromossomos.

4.4 Cruzamento de 2 pontos – Função *crossover2pontos*

Dentre os diferentes tipos de cruzamento discutidos na Seção 2.3, o escolhido para se utilizar no processo de evolução foi o Cruzamento de 2 pontos.

A função *crossover2pontos* (Figura 9) utiliza dois pais selecionados previamente na função *algoritmoGenetico* (Figura 9) e gera dois filhos que posteriormente substituirão os indivíduos de piores avaliações da população, utilizando-se para isso a estratégia *Steady State*.

Assim, selecionam-se aleatoriamente duas posições distintas entre 1 e o tamanho do cromossomo, que serão os pontos de corte dos cromossomos dos pais. Feito isso, realizam-se as trocas genéticas entre os pais e geram-se as codificações dos filhos (Figura 14).

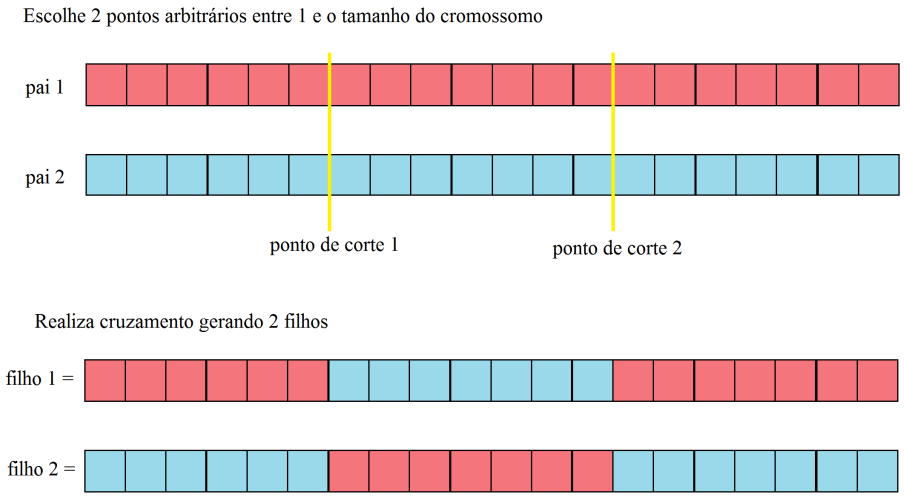

Figura 14: Diagrama de funcionamento do cruzamento de 2 pontos.

4.5 Estratégia de Evolução – Função $algoritmoGenetico$

O Algoritmo Genético $algoritmoGenetico$ (Figura 9) necessita dos seguintes parâmetros: número máximo de gerações $nger$; a quantidade de indivíduos $ncrom$; as dimensões da rede de genes: l linhas e c colunas; a taxa de mutação $taxaMut$; o nome dos arquivos de entrada e saída do circuito desejado $entradaTxt$ e $saidaTxt$, respectivamente; o vetor $portas$, que indica quais primitivas lógicas podem ser usadas para a construção do circuito lógico; o tipo de circuito (combinacional ou sequencial) e a quantidade de estados da máquina (se o circuito for apenas combinacional, a quantidade de estados é zero) definidos, respectivamente, em $tipo$ e $qntEstados$. O vetor de portas definido possui cinco posições, uma para cada função lógica, que pode conter 0 ou 1, indicando se aquela função está ou não habilitada (Tabela 4).

Tabela 4: Exemplo do vetor $portas$. Neste caso, a função XOR está desabilitada.

AND	OR	XOR	NOT	NOP
1	1	0	1	1

Inicialmente, gera-se a população inicial com características aleatórias, e então se começa o processo de evolução até que se atinja o número máximo de gerações ou que se encontre um indivíduo com erro (custo) nulo. A cada geração, realiza-se a decodificação e o cálculo de aptidão da população.

A estratégia de evolução utilizada na PGC desenvolvida mistura a Estratégia $\mu + \lambda$ e *Steady State*. Após se obter a avaliação de todos os indivíduos da população, é necessário escolher um cromossomo "pai" da geração. Para isso, seleciona-se o cromossomo com melhor avaliação da geração atual. Se sua avaliação for melhor ou igual à avaliação do pai da geração anterior, então ele passa a ser o novo pai da geração. Caso contrário, permanece o pai da geração anterior até que se obtenha um melhor ou igual.

Depois, aplica-se a mutação no pai, gerando-se todos os filhos que irão compor a próxima geração. Além da mutação, selecionam-se de forma aleatória três cromossomos, onde os dois melhores são designados para realizarem o cruzamento. O cruzamento produz dois filhos, que por sua vez substituem os dois indivíduos de piores avaliações da geração atual.

Por fim, incrementa-se o número de gerações passadas e o ciclo se repete para o cálculo de aptidão e aplicação dos operadores genéticos, até que se obtenha um indivíduo com valor zero de avaliação.

4.6 Fenótipo da solução – Função $plotFen$

Como até o momento o Algoritmo Genético tratou os circuitos lógicos como vetores de números inteiros, seguindo uma codificação da Programação Genética Cartesiana, é necessário transformá-los em ligações entre portas lógicas com o intuito de validar os resultados gerados.

A função $plotFen$ (Figura 9) realiza a representação gráfica do circuito lógico gerado e destaca os genes ativos (fenótipo) em vermelho. A função desenha no gráfico cartesiano as entradas, saídas do circuito e os genes do cromossomo. O cromossomo solução é lido gene por gene, enquanto as funções de cada gene são representadas por figuras na tela. Nesse processo, as coordenadas dos pontos de entradas e saídas dos genes são armazenadas em uma estrutura, para serem utilizadas posteriormente nas conexões entre os genes.

Depois de desenhado todo o cromossomo na tela, com todos os genes (genótipo completo), é preciso determinar o fenótipo, ou seja, os genes que efetivamente se interligam às saídas do circuito. Para isso, utiliza-se a função $destacaSolucao$ (Figura 9) que percorre de forma recursiva o caminho de volta de cada saída do circuito até as entradas. No caminho realizado, os genes percorridos são marcados com o valor 1 em um vetor $caminho$. Cada posição do vetor $caminho$ representa um gene do cromossomo. As posições contendo 1 indicam que aqueles genes são ativos e os que contém 0 são inativos.

A função $plotCaminho$ (Figura 9) se encarrega de varrer o vetor $caminho$ e substituir por figuras na cor vermelha os genes ativos indicados na tela. Ao final desse processo, obtém-se uma imagem como a apresentada na Figura 15.

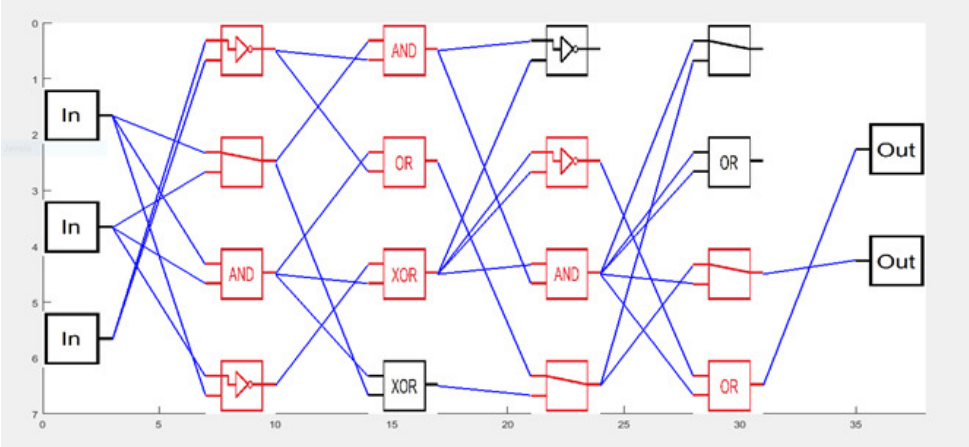

Figura 15: Fenótipo da solução para um circuito lógico de parâmetros $n = 3,\ s = 2,\ l = 4, c = 4$. As figuras *In* representam as entradas e *Out* representam as saídas do circuito.

5 Arquitetura Flexível em *Hardware*

A próxima etapa do projeto consistiu em transformar o circuito gerado pela PGC em implementação de *hardware* para reconfigurar uma FPGA.

Depois que o AG evolui até encontrar uma configuração de circuito que satisfaça a Tabela Verdade desejada, é necessário implementá-la utilizando uma arquitetura flexível e reconfigurável, neste caso, escrita em Verilog-HDL (VERILOG, 1996).

5.1 Configuração de Parâmetros

Primeiramente, o vetor de números que representa o cromossomo solução é convertido em arquivo de texto e os números inteiros em valores hexadecimais no MATLAB por meio da função $crom2txt$ (Figura 17a).

A função $crom2txt$ retorna também os parâmetros necessários a serem utilizados na arquitetura flexível. São eles: o número de entradas e saídas do circuito (n e s); as dimensões do grafo de nós (linhas n_r e colunas n_c, respectivamente); o número de *bits* $nBits$ necessários para representar todos os valores inteiros do cromossomo; o número de elementos do cromossomo $qntElementos$; o tipo do circuito (se combinacional, $tipo = 0$ e se sequencial $tipo = 1$); além da quantidade de *bits* necessários para representar a quantidade de estados da máquina de estados (ver Seção 6.3).

O número de *bits* é obtido por meio da representação binária do maior elemento inteiro do vetor. Se, por exemplo, o maior elemento é 58, a representação binária dele será: 111010, portanto, o número de *bits* será 6. Essa informação é necessária para realizar a leitura do cromossomo pela arquitetura flexível desenvolvida nesta proposta.

Os parâmetros então são informados manualmente nas declarações de parâmetros do módulo $circuitoCompleto$, em ordem na linguagem e podendo então sintetizar o *hardware*. A leitura é realizada por uma função nativa, escrita em Verilog-HDL, chamada $readmemh$, que carrega todo o arquivo de texto na matriz de *bits* $crom$, podendo depois ser facilmente manipulada pela linguagem (Figura 17b).

5.2 Estrutura de Células e Conexões

Os genes da PGC são representados por células lógicas em *hardware* (Figura 16) descritas em Verilog-HDL pelo módulo $celula$. Cada célula possui todos os operadores lógicos da Tabela de Funções (Tabela 2) implementados. O operador que será utilizado em cada célula é definido por um multiplexador controlado pelo seletor $selFun$, que se conecta ao primeiro elemento de cada gene na PGC, ou seja, o

índice da Tabela de Funções. Na figura, os seletores *selE1* e *selE2* selecionam as entradas de cada célula. Mais detalhes dessas conexões podem ser vistos na Figura 17.

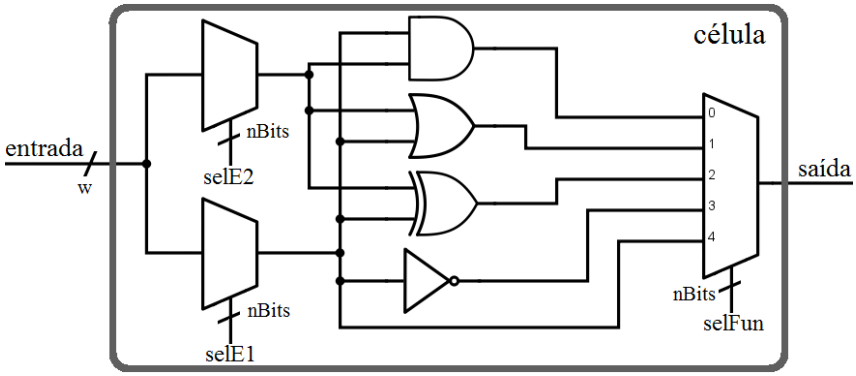

Figura 16: Representação da célula lógica utilizada na arquitetura flexível gerada por $celula$, tendo como entrada os barramentos $entrada$, $selFun$, $selE1$ e $selE2$, e um fio de saída. A largura w do barramento de entrada depende se a célula está conectada com as entradas do circuito (largura n) ou com o barramento $sCelula$ (largura $n_r \times n_c$).

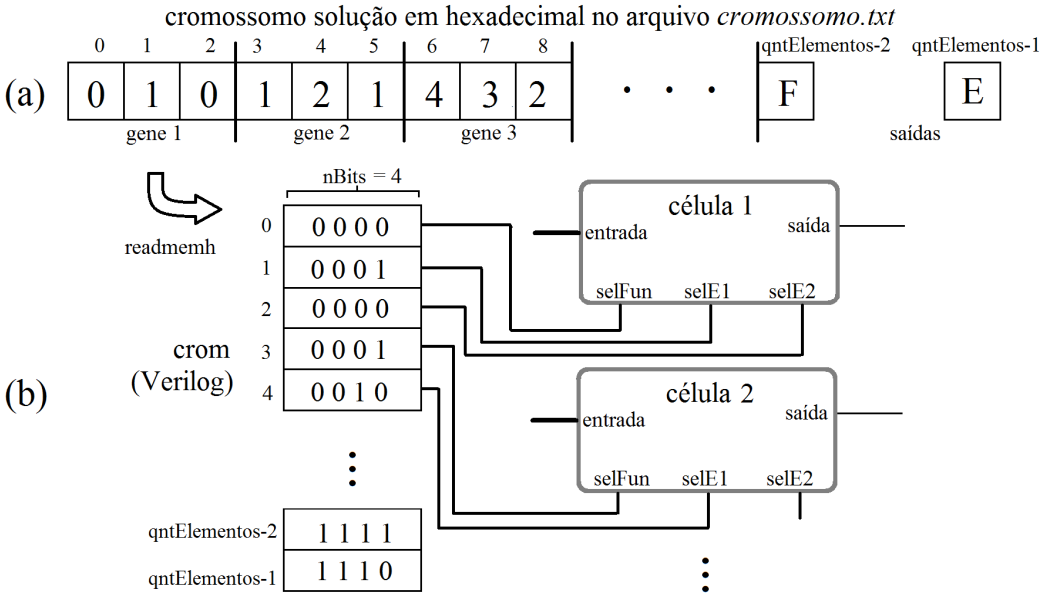

Figura 17: Detalhes de ligação do barramento de controle de cada célula. O arquivo gerado pela função $crom2txt$ contém o cromossomo solução em hexadecimal que é carregado para a matriz $crom$ pela função $readmemh$ em Verilog.

Nessa estrutura, todas as saídas das células são conectadas em um único barramento $sCelula$ de largura $n_r \times n_c$ (Figura 18), onde esse barramento se conecta também a todas as entradas das células, exceto às células da primeira coluna, que se conectam diretamente às entradas do circuito.

Assim, a interligação correta entre as células acontece por meio de multiplexadores implementados em Verilog-HDL. O módulo $geraRede$, descrito em Verilog-HDL, é responsável por essas ligações.

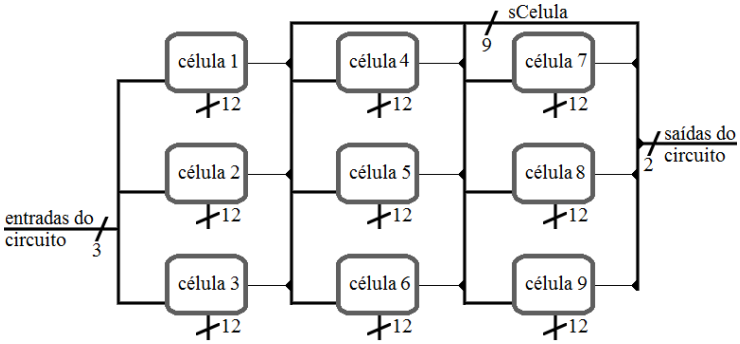

Figura 18: Representação de um exemplo de arquitetura gerada pelo módulo $geraRede$ utilizando $n = 3$, $s = 2$, $n_r = 3$, $n_c = 3$, $nBits = 4$ e $qntElements = 29$.

5.3 Representação e implementação em *hardware* de circuitos sequenciais

Os circuitos sequenciais tratados pela arquitetura referem-se à Máquinas de Estados Finitos de Moore. Nelas, os valores de saída dependem apenas do estado atual ao qual a máquina se encontra. A combinação da entrada da máquina e o estado atual determina qual será o próximo estado (MOORE, 1956).

A informação do estado atual é armazenada em elementos de memória. Na implementação corrente, foram utilizados *Flip-Flops* do Tipo D. O número de *Flip-Flops* utilizados depende do número de estados que a máquina possui. Por exemplo, uma máquina que possui 6 estados deverá ser projetada com 3 *Flip-Flops*, pois são necessários 3 *bits* para representar o valor 6 em binário – $(6)_{10} = (110)_2$, o que implica $nBitsE = 3$.

O circuito combinacional embutido na máquina é utilizado para determinar o valor de saída em função do estado atual e também para determinar o próximo estado em função do estado atual e do valor de entrada corrente.

Assim, a máquina, de maneira genérica, é formada por um circuito combinacional que tem como entradas as entradas da máquina de estados e o estado atual. As saídas do circuito combinacional são as

saídas da máquina e o estado futuro. Os *Flip-Flops* realimentam o circuito de forma que recebam como entrada o estado futuro e enviem como saída o estado atual (Figura 19).

A comutação de estados ocorre de forma síncrona devido aos *Flip-Flops* do Tipo D repassarem o valor da entrada para a saída a cada borda de subida do *clock*. Dessa forma, eles têm o papel de controlar o fluxo de realimentação do circuito.

O estado inicial da máquina é definido pelo pino $reset$. Toda vez que a máquina é reiniciada, os *Flip-Flops* carregam o estado atual com valor 0, que representa o estado inicial da operação.

A criação do *hardware* completo, tanto do circuito combinacional puro quanto da máquina de estados contendo a parte combinacional e *Flip-Flops*, é feita por meio da função, em Verilog-HDL, $circuitoCompleto$.

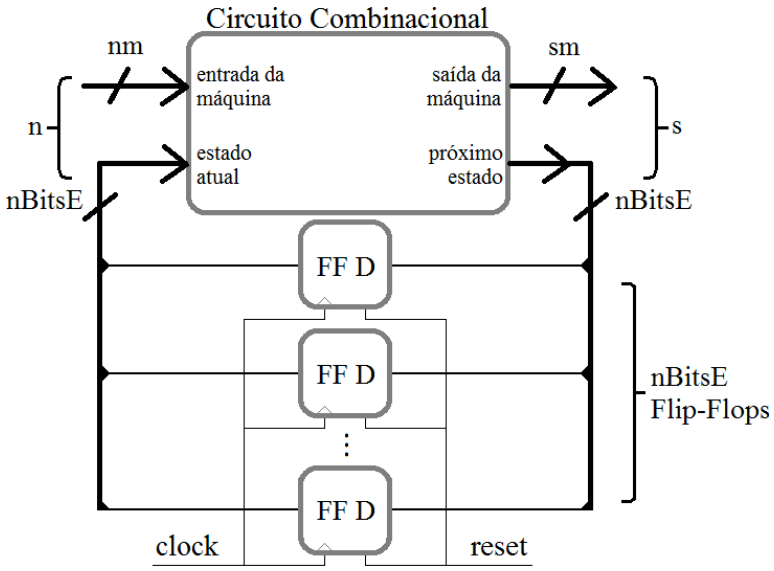

Figura 19: Estrutura geral do *hardware* gerado para a Máquina de Estados por meio do módulo completo $circuitoCompleto$, onde nm representa o número de entradas e sm o número de saídas da referida Máquina. Os índices n e s representam, respectivamente, o número de entradas e saídas do circuito combinacional embutido na máquina, sendo $n = nm + nBitsE$ e $s = sm + nBitsE$.

Na Figura 20, mostra-se um exemplo de Máquina de Estado identificadora de uma dada seqüência e sua respectiva Tabela Verdade para as saídas e as transições de estados (Tabela 5) referidas. As entradas e saídas do circuito combinacional devem ser informadas ao AE em MATLAB, que irá evoluir um circuito combinacional para posteriormente ser utilizado na geração da arquitetura flexível sobredita.

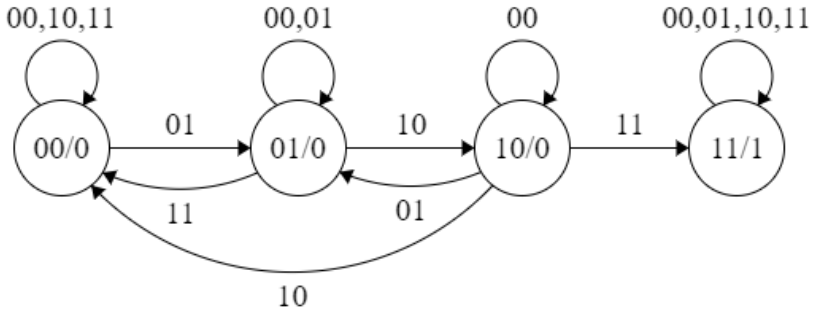

Figura 20: Máquina de Estados identificadora de seqüências. Se a seqüência inserida na ordem for 00,01,10,11, neste exemplo, ela gerará o *bit* 1 na saída, e para qualquer outra combinação de seqüência, gerará 0.

Tabela 5: Tabela de Transições da Máquina de Estados identificadora de seqüência.

Entradas do Circuito Combinacional (n)					**Saídas do Circuito Combinacional (s)**		
Estado Atual ($nBitsE$)		**Entradas da Máquina (nm)**			**Próximo Estado ($nBitsE$)**		**Saídas da Máquina (sm)**
0	0	0	0		0	0	0
0	0	0	1		0	1	0
0	0	1	0		0	0	0
0	0	1	1		0	0	0
0	1	0	0		0	1	0
0	1	0	1		0	1	0
0	1	1	0		1	0	0
0	1	1	1		0	0	0
1	0	0	0		1	0	0
1	0	0	1		0	1	0
1	0	1	0		0	0	0
1	0	1	1		1	1	0
1	1	0	0		1	1	1
1	1	0	1		1	1	1
1	1	1	0		1	1	1
1	1	1	1		1	1	1

6 *Toolbox* GPLAB

A *toolbox* GPLAB para MATLAB foi utilizada nos estudos para comparação dos resultados obtidos com o Algoritmo de Programação Genética Cartesiana desenvolvido neste TCC. Apesar da GPLAB não utilizar a estrutura cartesiana, sua utilização tem servido para fins de estudo e compreensão da parte de operadores genéticos, haja vista que seus resultados têm se mostrado bastante satisfatórios.

Existem três modos de operação da *toolbox*: o leigo, o regular e o avançado (SILVA, 2007). No modo leigo, o usuário apenas precisa informar a quantidade máxima de gerações e o número de indivíduos da população, os demais parâmetros são definidos com valores padrões; o usuário regular é capaz de definir outros parâmetros, como o modo de operação do Programa Genético (regressão simbólica ou *artificial ant* – usada em problemas envolvendo rotas), funções utilizadas nos cromossomos, operadores genéticos, estratégias de evolução e métodos de seleção. O usuário avançado é capaz de definir e incluir novos operadores genéticos, modos de operação e métodos de seleção, além de novas funções.

Todas as suas variáveis e parâmetros atribuídos que determinam o modo do algoritmo executar são armazenadas em uma estrutura chamada $params$. Quando configurado para otimização de funções e regressão simbólica, é necessário informar as entradas e as saídas desejadas da função. Se o problema for do tipo *artificial ant*, é necessário então entrar com a trilha de comida em forma de uma matriz binária e em seguida com a quantidade de cápsulas de comida disponíveis na trilha.

A GPLAB trabalha com populações de tamanho variado, e que crescem dinamicamente. Para isso, existem alguns parâmetros que devem ser definidos e que restringem o tamanho das árvores geradas. Em $initmaxlevel$, define-se o tamanho máximo (quantidade de níveis) que novas árvores podem ter. Além disso, existem três métodos de crescimento dinâmico que podem ser configurados em $initipoptype$: *fullinit*, *growinit* e *rampedinit*.

Em *fullinit* os nós da árvore são criados como nós internos até que a altura máxima (*initimaxlevel*) seja atingida e no último nível eles são nós-folha. Desta forma, a árvore gerada é perfeitamente balanceada. O método *growinit* cria aleatoriamente nós internos e nós-folha, possibilitando que sejam criadas árvores muito desbalanceadas. E no método *rampedinit*, metade são criados com o método *fullinit* e a outra metade com *growinit*. Com isso, há grande diversidade de indivíduos na população, visto que agora se tem árvores perfeitamente balanceadas e árvores desbalanceadas.

A variação de tamanho dos indivíduos proporcionada pela estrutura de árvores geradas na GPLAB é análoga à utilização do operador NOP no Algoritmo de Programação Genética Cartesiana proposto neste trabalho. A Figura 21 demonstra um exemplo de estrutura de árvore gerada pela *toolbox* que representa um circuito equivalente para a porta lógica XOR. Os nós denominados “X1” e “X2” representam as entradas do circuito e a saída é o resultado obtido na raiz da árvore.

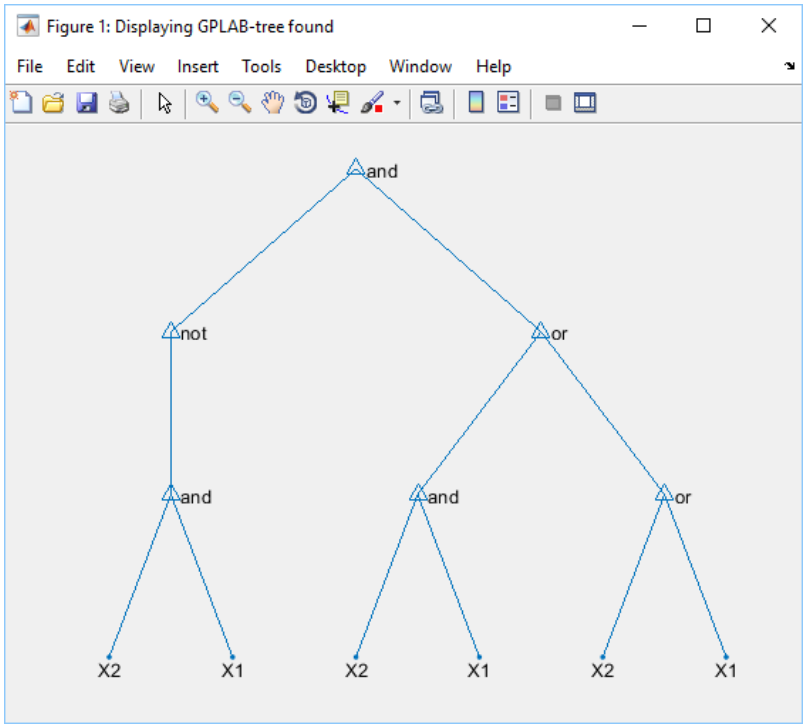

Figura 21: Circuito equivalente à porta XOR gerado em forma de árvore pela GPLAB.

6.1 Configuração de parâmetros

Para a GPLAB funcionar corretamente, gerando configurações de circuitos desejados, é necessário definir alguns parâmetros, como no exemplo dado a seguir para a geração de um circuito equivalente XOR:

```
1 % Define valores padrões nos parâmetros
2 p=resetparams;
3 % setoperators(parametros, 'nome do operador', pais requeridos, filhos gerados)
4 p=setoperators(p,'crossover',2,2,'mutation',1,1);
5 % Define quais funções serão usadas. setfunctions(parametros, 'nome da função',
argumentos)
6 p=setfunctions(p,'and',2,'or',2,'not',1);
7 p.initipoptype='fullinit';
8 % Entradas do circuito
9 p.datafilex='inputXOR.txt';
10 p.datafiley='outputXOR.txt';
11 % Execução
12 [dados,melhorIndividuo]=gplab(500,50,p);
13 % Desenha árvore solução e guarda em arquivo tree.txt
14 drawtree2(melhorIndividuo.tree);
```

Primeiramente, define-se uma estrutura que conterá todos os parâmetros (nesse caso, p será a estrutura de parâmetros). Com a função $resetparams$ a estrutura recebe todos os valores iniciais padrão. Depois disso, é necessário definir quais operadores a GPLAB irá utilizar para construir os circuitos. A função $setfunctions$ é utilizada para definir quais operadores serão utilizados, onde se informa para cada operador o seu nome e quantas entradas são necessárias para a operação, respectivamente.

A seguir, define-se o tipo de crescimento dinâmico com a função $initipoptype$. Além disso, é necessário informar quais são as entradas e as saídas do circuito, ambas contidas em arquivos de texto (no exemplo, "*inputXOR.txt"* contém os valores de entrada e "*outputXOR.txt"* os de saída) e informadas nos parâmetros $datafilex$ e $datafiley$, respectivamente.

Para iniciar a busca, basta chamar a função $gplab$ indicando, na ordem, o número máximo de gerações, a quantidade de indivíduos da população e a estrutura de parâmetros. Quando encontrada uma solução, a função retorna em forma de estrutura todos os dados obtidos durante a execução do Algoritmo Genético em $dados$ e todas as informações da estrutura de árvore do indivíduo solução em $melhorIndividuo$. Por fim, pode-se gerar o fenótipo da solução com a função $drawtree$, passando-se como parâmetro a estrutura de árvore que se deseja representar graficamente.

6.2 Conversão da estrutura de Árvore em Rede Cartesiana

Para implementar a configuração de circuito em *hardware* na FPGA, é necessário transformar a estrutura de árvore na forma cartesiana para que possa ser mapeada pela arquitetura flexível em Verilog-HDL detalhada no Capítulo 6.

A função $tree2cartesian$ recebe a árvore gerada pela GPLAB em forma de vetor da seguinte maneira: aproveitando a função $drawtree$ já implementada na *toolbox* que percorre a árvore em *pos-order* (primeiro os nós e por último a raiz da sub-árvore), é feita uma modificação para que os nós sejam postos em um arquivo de texto ($drawtreeModificada$). Depois de gerado o arquivo de texto contendo todos os nós, este é facilmente convertido em um vetor de caracteres por meio da função $tree2cell$. A árvore representada na Figura 21 é convertida em um vetor de caracteres conforme a Tabela 6.

Tabela 6: Árvore representada em *pos-order* por vetor de caracteres

X2	X1	and	not	X2	X1	and	X2	X1	or	or	and

A função $tree2cartesian$ se encarrega de transformar o vetor de caracteres que representa a árvore em um vetor que configura corretamente a rede cartesiana.

O primeiro passo é identificar quantas linhas e colunas na rede cartesiana serão necessárias para representar a árvore. A quantidade de colunas é igual ao número de níveis da árvore menos um e a quantidade de linhas obtém-se pela quantidade de nós pertencentes ao penúltimo nível. Para transformar a árvore da Figura 21 em rede uma cartesiana, por exemplo, seria necessária uma rede de 3 colunas por 3 linhas.

Depois de definida a dimensão da rede, usa-se a função $geraPop$ do AE para gerar um vetor preenchido de operadores NOP. Dessa forma, os nós correspondentes na árvore são sobrepostos e os demais permanecem inutilizáveis.

O passo seguinte é indexar cada nó da árvore e convertê-los na numeração utilizada na configuração cartesiana. Primeiramente, trocam-se os valores dos nós referentes às entradas do circuito (X1 e X2) por índices 1 e 2. Depois, para cada conjunto de operador lógico e entradas da árvore é atribuído um índice de gene, ao mesmo tempo em que é convertido em gene no vetor cartesiano (três posições de vetor: função e duas entradas). Essa varredura é feita para cada nível da árvore e os índices de genes dos níveis anteriores servem de entrada para os nós do nível atual da varredura.

Nesse processo, a varredura começa do último e vai até o primeiro nível, sendo que cada um representa uma coluna na rede cartesiana. Definiu-se por padrão que genes onde o operador lógico é a porta NOT, a segunda entrada é igual a primeira, já que esta não possui efeito sobre o genótipo da solução. A Figura 22 mostra um exemplo do processo de modo geral.

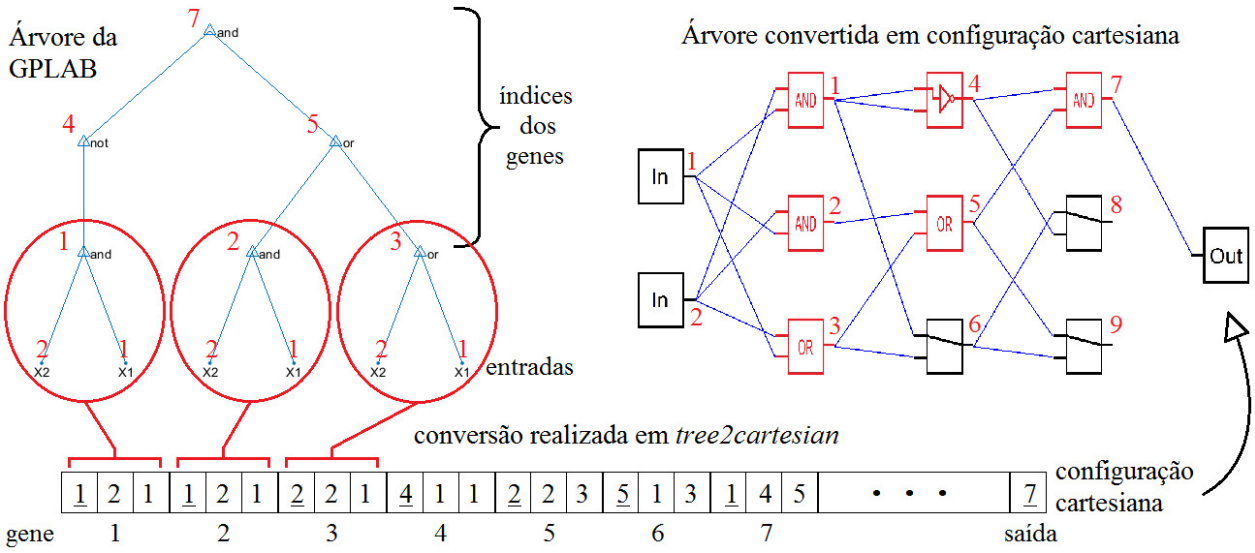

Figura 22: Exemplo de conversão de árvore da GPLAB em configuração cartesiana realizada pela função *tree2cartesian*

De forma geral, o processo de conversão de árvore para estrutura cartesiana segue a seguinte organização, conforme a Figura 23.

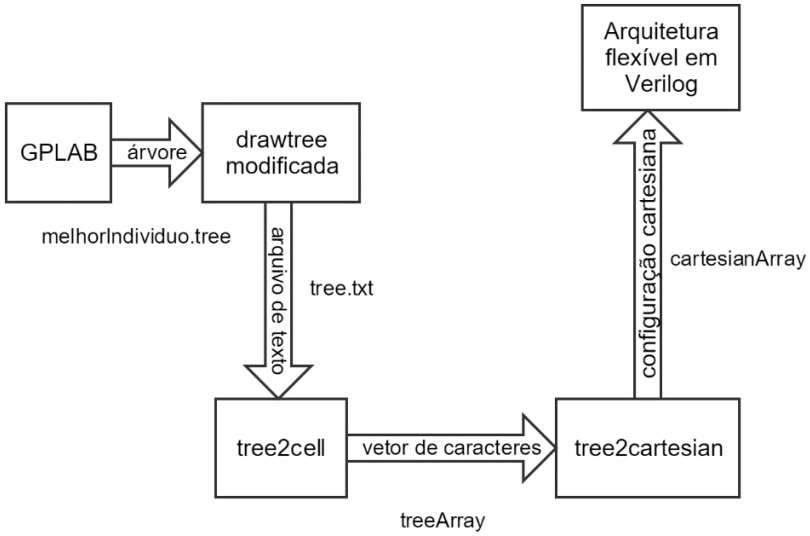

Figura 23: Visão geral do esquema de conversão. Partindo da GPLAB até a Arquitetura flexível em *Verilog*.

7 Resultados obtidos

7.1 Circuitos combinacionais

Para se obter alguns resultados de execução do Algoritmo Evolutivo desenvolvido, utilizaram-se três 3 valores distintos para os parâmetros de taxa de mutação e tamanho da rede de nós, conforme descrito a seguir, para alguns tipos de circuitos combinacionais encontrados na literatura.

Em cada uma das permutações de parâmetros, executou-se o Algoritmo Evolutivo 30 vezes e anotou-se, para cada uma delas, a geração na qual foi encontrada uma solução satisfatória e o respectivo cromossomo resultante.

Os valores utilizados para cada taxa de mutação testada foram: 10%, 30% e 50%; e as dimensões da rede de nós foram: 4×4, 6×6 e 8×8. Para o teste do Circuito XOR, utilizou-se o limite de 500 para o número máximo de gerações necessárias para convergir a uma solução, e foram utilizadas apenas primitivas AND, OR, NOT e NOP. O circuito verificador de paridade de 2 *bits* foi testado com 10 indivíduos. Os circuitos *Half Adder, Full Adder* e *Multiplier* de 2 *bits* utilizaram uma população de 5 indivíduos, e para todos os casos, o número máximo de gerações foi igual a 10000. Execuções que não conseguiram convergir para uma solução adequada antes de se atingir o número máximo de gerações foram consideradas sem sucesso.

7.1.1 Caso de estudo 1 - Circuito XOR

O circuito equivalente da porta XOR, obtido na literatura, pode ser visto na Figura 24:

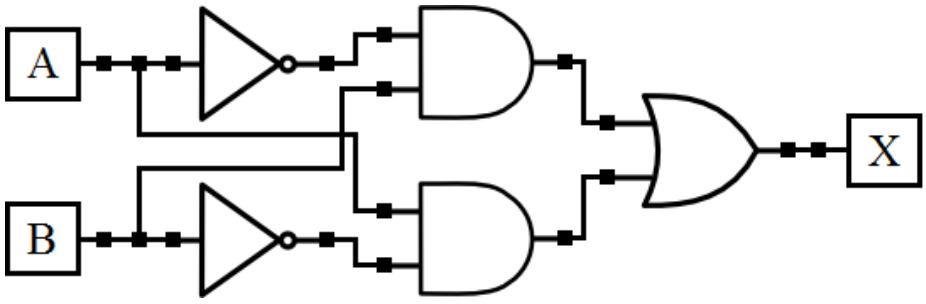

Figura 24: Circuito equivalente à porta lógica XOR.

Após a execução dos testes, conforme descrito anteriormente, os resultados obtidos pelo sistema evolutivo desenvolvido para a geração automática do circuito XOR (Tabela 7) estão resumidos na Tabela 8.

Tabela 7: Tabela Verdade do Circuito XOR utilizada como entrada do Caso de Estudo 1.

A	B	X
0	0	0
0	1	1
1	0	1
1	1	0

Analisando-se os dados obtidos (Tabela 8), pode-se concluir que para gerar um circuito equivalente à porta XOR, os melhores resultados foram obtidos para dimensões maiores da rede de nós. Neste caso, o Algoritmo Evolutivo convergiu para uma solução mais rapidamente com uma rede de dimensões 8×8. Observa-se também que a taxa de mutação não teve tanta influência no desempenho do AE, de acordo com as médias e desvios padrão encontrados para este caso. Em todos os casos, o AE obteve 100% de sucesso nas execuções, ou seja, para cada configuração de parâmetros, conseguiu-se um resultado em menos de 500 gerações.

O tamanho médio do fenótipo gerado é indicado pela quantidade de nós ativos, visto que esses nós são aqueles que se conectam de alguma forma às saídas do circuito e influenciam nos valores gerados.

Tabela 8: Resumo dos resultados para Circuito XOR.

Dimensões	Taxa de Mutação	Média (Gerações)	Desvio Padrão σ	Nós ativos	Sucesso	Índice
4×4	10%	33	39	8	100%	1
	30%	13	12	8	100%	2
	50%	11	8	8	100%	3
6×6	10%	10	10	14	100%	4
	30%	6	5	13	100%	5
	50%	7	7	15	100%	6
8×8	10%	7	7	21	100%	7
	30%	6	5	21	100%	8
	50%	5	6	23	100%	9

Na Tabela 9 são mostrados os resultados obtidos para 30 execuções do AG da *toolbox* GPLAB. Não foi possível variar a taxa de mutação neste caso, porque a GPLAB utiliza uma técnica de mutação

otimizada que aumenta ou diminui ao longo do tempo. Os parâmetros foram configurados de acordo com a Seção 7.1 (primitivas AND, OR, NOT, NOP e $initpoptype = rampedinit$).

Tabela 9: Resumo dos resultados para Circuito XOR utilizando GPLAB para 30 execuções do algoritmo.

Média (Gerações)	Desvio Padrão σ
94	97

Na Figura 25 são apresentados os resultados obtidos para a execução do Circuito XOR usando diferentes Estratégias de Evolução, dados os parâmetros da Tabela 8. A Estratégia Atual (descrita na Seção 5.5) mostrou-se mais eficiente para esta aplicação. A estratégia de Mutação Casual considera uma taxa de mutação para cada indivíduo selecionado. Neste caso, decide-se previamente o número de indivíduos que irão sofrer mutação. Na estratégia atual, todos os indivíduos sofrem mutação e a taxa de mutação é aplicada sobre os genes de cada indivíduo. Por exemplo, para uma taxa de mutação de 10%, na Estratégia Atual, 10% dos genes de todos os indivíduos sofrem mutação. Na Mutação Casual, 10% dos indivíduos são selecionados para sofrer mutação e 10% de seus genes são efetivamente mutados.

A Mutação Pontual marca os nós ativos (fenótipo) do cromossomo e realiza mutação em genes aleatórios. Também foi experimentada a estrutura proposta por Miller (2011) – a principal diferença é que as saídas do circuito podem se conectar com nós de qualquer nível do grafo – variando-se o *level-back* de 1 até o número de colunas, ou seja, todos os nós podem se conectar com nós de colunas anteriores.

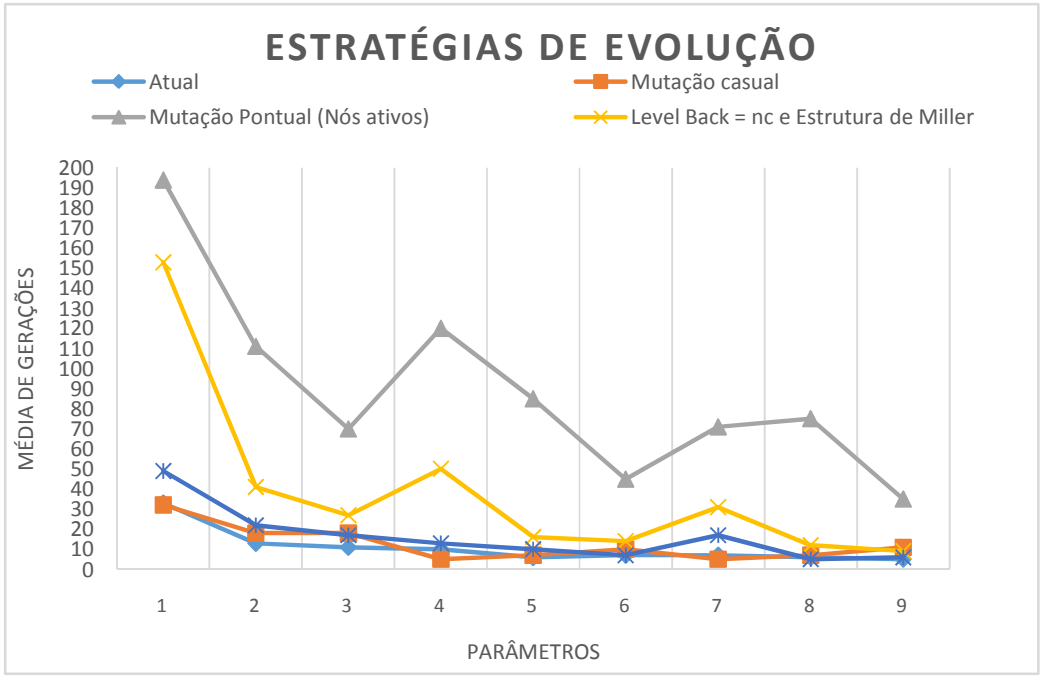

Figura 25: Gráfico parâmetros × média de gerações para diferentes Estratégias de Evolução.

7.1.2 Circuito verificador de paridade ímpar de 3 *bits*

O circuito verificador de paridade ímpar de 3 *bits* de entrada é representado pela Tabela Verdade dada a seguir (Tabela 10), e seu circuito digital equivalente, utilizando apenas portas AND, OR e NOT, pode ser visto na Figura 26.

Tabela 10: Circuito verificador de paridade ímpar de 3 *bits*. A,B,C são entradas e X é saída.

A	B	C	X
0	0	0	1
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	0

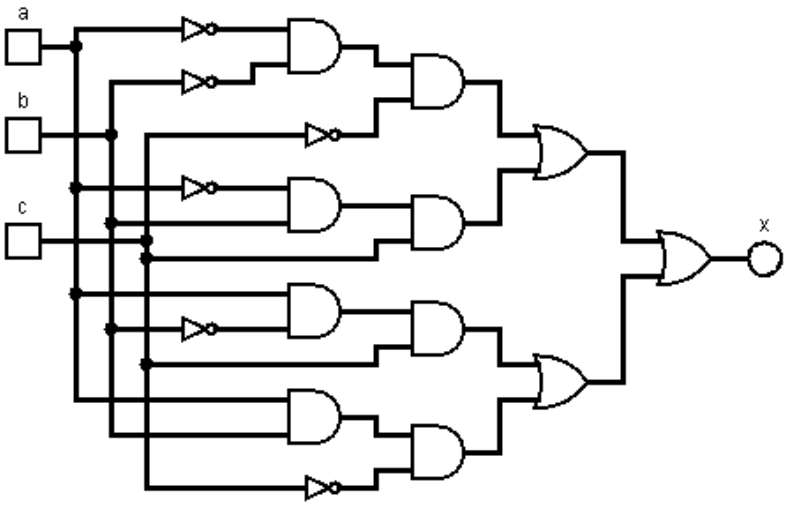

Figura 26: Circuito lógico verificador de Paridade ímpar de 3 *bits* gerado a partir da técnica de minimização utilizando Mapa de Karnaugh para a Tabela Verdade apresentada na Tabela 10.

Os testes executados para este caso de estudo utilizaram uma população de 10 indivíduos, no qual o número máximo de gerações foi limitado em 10000. As primitivas utilizadas neste exemplo foram: AND, OR, NOT e NOP.

Por se tratar de um problema mais complexo, optou-se por realizar testes apenas com as dimensões de grafos 6×6 e 8×8. Além disso, observou-se que para taxas de mutação acima de 30%, o AE não obteve sucesso nas execuções, visto que tal quantidade de mutação faz com que a população, de maneira geral, seja muito dispersa no espaço de busca, dificultando-se o processo de encontrar uma solução em menos de 10000 gerações. Observa-se que o melhor desempenho ocorreu quando para uma rede cartesiana maior que 8×8. A Tabela 11 contém o resumo dos resultados obtidos neste caso.

Os resultados obtidos também foram comparados com os encontrados por Walker e Miller (2008), por Programação Genética Cartesiana, por meio das funções: AND, OR, NAND e NOR.

Tabela 11: Resultados obtidos para o circuito verificador de paridade par de 3 *bits* após 30 execuções.

Parâmetros		**Média de gerações**	**Desvio Padrão σ**	**Nós Ativos**	**Sucesso**	**Walker e Miller (2008)**
Taxa de Mutação: 10% População: 10 Gerações: 10000	6×6	3489	2974	18	93%	5993
	8×8	2162	1541	27	100%	

7.1.3 Caso de estudo 2 – Circuito *Half Adder*

O circuito *Half Adder* (Figura 27) – meio somador binário – possui a Tabela Verdade descrita na Tabela 12. Neste caso, os testes foram executados 30 vezes para as seguintes taxas de mutação: 1%, 10% e 30%; também, as dimensões adotadas para o referido problema foram: $4 \times 4, 6 \times 6$ e 8×8; em todos os casos, foram anotadas as médias de gerações usadas para encontrar as soluções desejadas (Tabela 13). Os testes foram realizados com uma população de 5 indivíduos e com o limite máximo de gerações fixado em 10000.

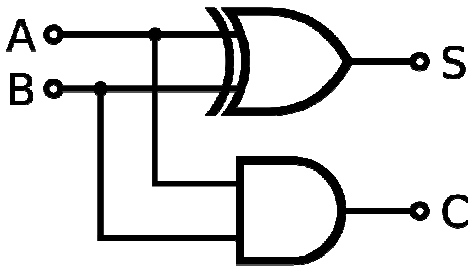

Figura 27: Circuito lógico *Half Adder*.

Tabela 12: Tabela Verdade do circuito *Half Adder*.

A	B	S	C
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Tabela 13: Resultados obtidos para o circuito *Half Adder* após 30 execuções.

Dimensões	Taxa de Mutação	Média	Desvio Padrão σ	Sucesso
4×4	1%	418	435	100%
	10%	57	56	100%
	30%	87	108	100%
6×6	1%	274	242	100%
	10%	67	55	100%
	30%	189	215	100%
8×8	1%	166	161	100%
	10%	65	62	100%
	30%	314	371	100%

Observa-se que para todos os tamanhos de rede cartesiana, os melhores resultados foram obtidos para uma taxa de mutação de 10%. Para uma taxa de mutação muito baixa (1%), há uma demora maior para a população convergir a uma solução satisfatório, pois, as mudanças aleatórias ocorrem mais lentamente. Em contrapartida, para uma taxa maior (30%), a tendência do AE é fazer buscas em direções aleatórias do espaço de soluções, assim como um Algoritmo de Força Bruta, conforme discutido no Capítulo 2.

7.1.4 Caso de estudo 3 – Circuito *Full Adder*

O circuito *Full Adder* (Figura 28) – somador binário completo de 2 *bits* – possui a Tabela Verdade descrita na Tabela 14. Assim como no caso anterior, os testes foram executados 30 vezes para as taxas de mutação: 1%, 10% e 30%; e para as dimensões: $4 \times 4, 6 \times 6$ e 8×8. Por fim, anotaram-se as médias de gerações usadas para encontrar as soluções em cada caso (Tabela 15). Os testes foram realizados com uma população de 5 indivíduos e com um limite máximo de 10000 gerações, utilizando-se as seguintes primitivas lógicas: AND, OR, XOR, NOT e NOP.

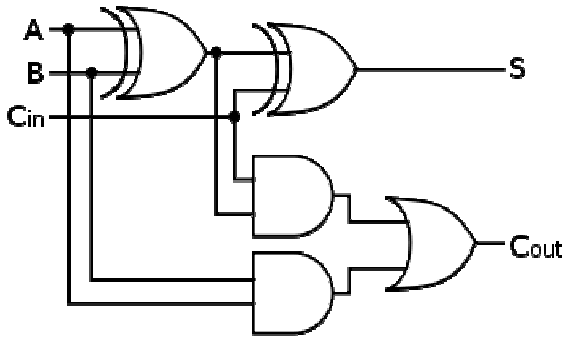

Figura 28: Circuito lógico *Full Adder*.

Tabela 14: Tabela Verdade do circuito *Full Adder*.

A	B	Cin	S	Cout
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Tabela 15: Resultados obtidos para o circuito *Full Adder* após 30 execuções.

Dimensões	Taxa de Mutação	Média	Desvio Padrão σ	Sucesso
4×4	1%	6546	3160	67%
	10%	1451	1301	100%
	30%	5458	3599	70%
6×6	1%	2788	2884	90%
	10%	1315	1345	100%
	30%	8275	3098	33%
8×8	1%	1193	1238	100%
	10%	2576	2075	97%
	30%	9861	763	3%

Observa-se que os melhores resultados, de maneira geral, foram obtidos utilizando uma taxa de mutação de 10% para todos os tamanhos de rede cartesiana, onde as taxas de sucesso foram de 100% ou muito próximo disso.

7.1.5 Caso de estudo 4 – Circuito *Multiplier 2 bits*

O circuito *Multiplier* de *2 bits* (Figura 29) – multiplicador binário de 2 *bits*, é mais complexo que os demais, sendo assim, com base nos resultados obtidos anteriormente, foi utilizada apenas uma taxa de mutação de 1% e uma rede cartesiana de dimensões 8×8. Os resultados encontrados são mostrados na Tabela 16.

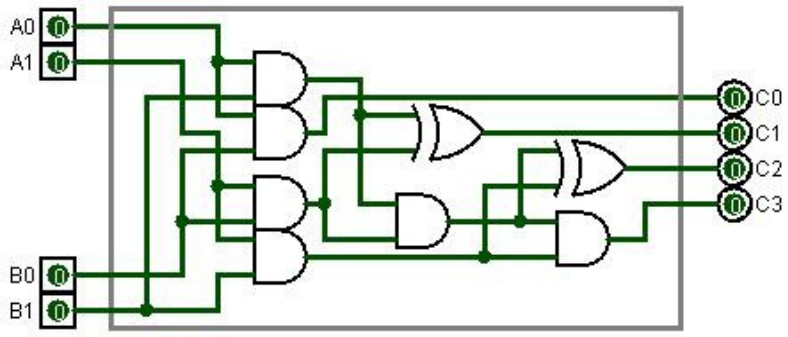

Figura 29: Circuito lógico *Multiplier* de *2 bits*.

Tabela 16: Resultados obtidos para o circuito *Full Adder* após 30 execuções.

Parâmetros		**Média de gerações**	**Desvio Padrão σ**	**Sucesso**
Taxa de Mutação: 1% População: 5 Gerações: 10000	8×8	8583	2313	33%

Observa-se que para esse tipo de circuito o AG não conseguiu bons resultados utilizando 10000 gerações.

7.2 Circuito Seqüencial

A máquina de estados utilizada para este estudo (Figura 30) é equivalente a um circuito verificador de paridade, onde a saída é 1 para número ímpar de *bits 1* inseridos e 0 para número par de *bits 1*. A Tabela de Estados encontra-se na Tabela 17. Para os resultados, variou-se a taxa de mutação de: 1%, 10% e 30%; e as seguintes dimensões foram utilizadas: 4×4, 6×6 e 8×8; Os resultados obtidos podem ser vistos na Tabela 18. Em seguida, a máquina foi implementada em FPGA utilizando a arquitetura flexível descrita no Capítulo 6 (Figura 31). As formas de onda obtidas para o circuito (Figura 32), confirmam os resultados da Tabela de Estados (17).

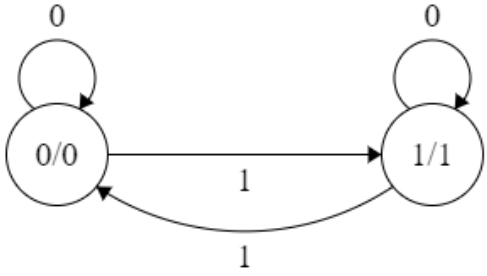

Figura 30: Máquina de estados verificadora de paridade.

Tabela 17: Tabela de Estados.

Estado Atual	Entrada da Máquina	Próximo Estado	Saída da Máquina
0	0	0	0
0	1	1	0
1	0	1	1
1	1	0	1

Tabela 18: Resultados obtidos.

Dimensões	Taxa de Mutação	Média	Desvio Padrão σ	Sucesso
4 × 4	1%	363	497	100%
	10%	86	61	100%
	30%	67	44	100%
6 × 6	1%	323	272	100%
	10%	67	54	100%
	30%	333	260	100%
8 × 8	1%	260	231	100%
	10%	149	116	100%
	30%	394	325	100%

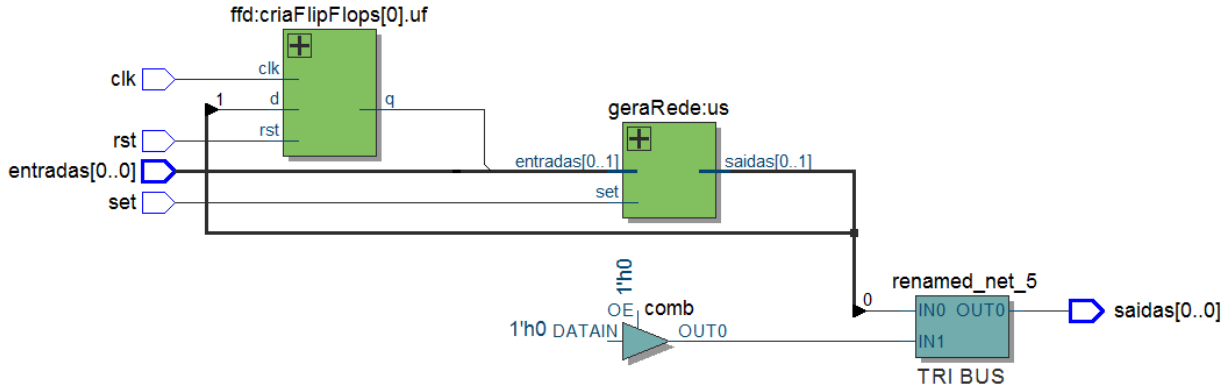

Figura 31: Arquitetura gerada para a Máquina de Estados em estudo. Gerado pelo visualizador RTL do Quartus II.

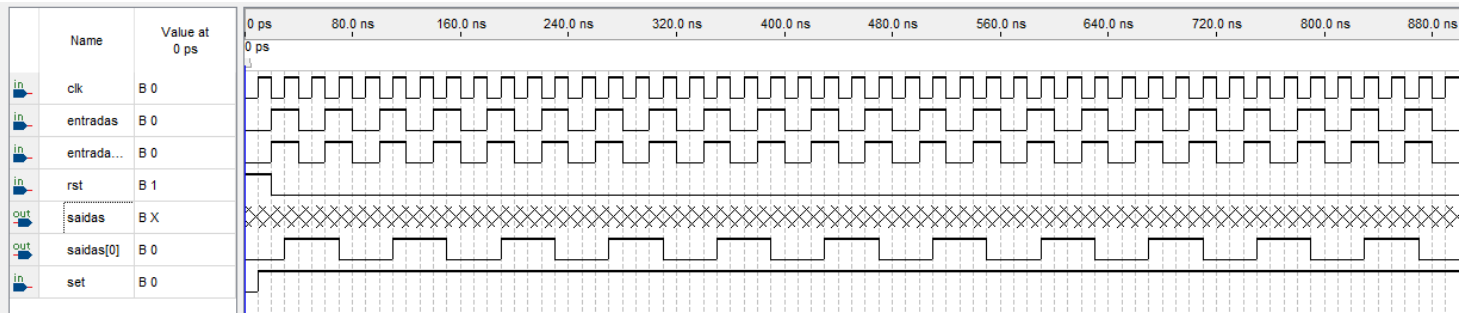

Figura 32: Simulação de onda gerada para Máquina de Estados em estudo. Gerada pelo simulador do Quartus II.

Observa-se que quando o número de *bits 1* inseridos na máquina é ímpar, a saída é 1, e quando o número é par, a saída é 0. Isso demonstra que a arquitetura gerada é válida e condiz com a Máquina de Estados em estudo.

8 Conclusões

O projeto apresentado contemplou todas as etapas propostas no cronograma inicial deste TCC, no qual foi realizado um estudo sobre Algoritmos Genéticos e Programação Genética Cartesiana, além de outros métodos inteligentes existentes na literatura. Também, foi desenvolvido um Algoritmo Evolutivo em MATLAB utilizando a técnica de Programação Genética Cartesiana para geração de circuitos combinacionais e sequenciais. Além disso, foi estudada a *Toolbox* GPLAB, e feita a conversão de sua estrutura de árvore para a estrutura em PGC utilizada, sendo esta também uma contribuição deste TCC. Os testes realizados se mostraram bem-sucedidos para diferentes casos de circuitos testados e foram comparados a outros presentes na literatura, mostrando assim a viabilidade da técnica proposta (SHANTI; CHEENAI, 2005; WALKER; MILLER, 2008; KAZARLIS; KALOMIROS; KALAITZIS, 2015; IRFAN et al., 2010; SEKANINA L; VASICEK, 2015).

A segunda parte do projeto consistiu em gerar uma arquitetura flexível em Verilog-HDL, para a implementação das soluções obtidas, por meio dos circuitos gerados no MATLAB, em FPGA. Portanto, tal arquitetura é capaz de gerar circuitos combinacionais e sequencias, de acordo com os resultados obtidos neste trabalho.

O diferencial do projeto corrente em relação aos trabalhos de mesmo contexto existentes na literatura, é a geração automática de *hardware*, e além disso, a utilização de PGC para a geração de circuitos sequenciais (Máquinas de Estado), haja vista que há poucos trabalhos realizados nesta área (ASHA, 2015; SOLEIMANI et al., 2011). Por fim, uma das dificuldades encontradas pela abordagem de evolução proposta neste TCC, corresponde ao tempo de treinamento do algoritmo evolutivo para encontrar uma solução satisfatória para circuitos mais complexos, assim, como trabalho futuro é pretendido desenvolver tal treinamento por meio de um *hardware* dedicado, com o intuito de acelerar o processamento do sistema durante a sua fase de treinamento.

REFERÊNCIAS

ASHA, S.; HEMAMALINI, R. R. **Synthesis of Adder Circuit Using Cartesian Genetic Programming**. Middle-east Journal of Scientific Research**,** Chenaai, India, p. 1181-1186. 2015.

ALTERA, Corp. **Altera's Development and Education Board**. Disponível em: < https://www.altera.com/solutions/partners/partner-profile/terasic-inc-/board/altera-de2-115-development-and-education-board.html >. Acesso em: 17 fev. 2016.

CELES, W.; CERQUEIRA R.; RANGEL J. L. **Introdução à Estrutura de Dados**. EditoraCampus, 2004.

DARWIN, C. H. **The Origin of Species, by means of Natural Selection**. London: John Murray, 1859.

DAVIS, L. D. **Handbook of Genetic Algorithms**. Van Nostrand Reinhold, 1991.

EIBEN, A. E.; SMITH, J. E. **Introduction to Evolutionary Computing**. Stringer, 2003.

GOLDBERG, D. E. **Genetic Algorithms in Search, Optimization and Machine Learning**. 1 ed. Boston: Addison-Wesley Longman Publishing Co., Inc., 1989.

GOMES, A. M. D. **O Problema do Caixeiro Viajante**. Disponível em: <http://www.uff.br/sintoniamatematica/grandestemaseproblemas/grandestemaseproblemas-html/audio-caixeiro-br.html> Acesso em: 15 jul. 2015.

HARDING, S. L.; MILLER J. F.; BANZHAF, W. **Self Modifying Cartesian Genetic Programming: Parity**. Proceedings of Congress on Evolutionary Computation, IEEE Press, p. 285-292. 2009.

HOLLAND, J. H. **Adaptation in Natural and Artificial Systems**. University of Michigan Press, Ann Arbor, Michigan; re-issued by MIT Press, 1992.

IRFAN, M. et al. **Combinational digital circuit synthesis using Cartesian Genetic Programming from a NAND gate template**. 2010 6th International Conference On Emerging Technologies (icet), [s.l.], p. 343-347, out. 2010. Institute of Electrical & Electronics Engineers (IEEE). DOI: 10.1109/icet.2010.5638462.

KARNAUGH, M. **The Map Method for Synthesis of Combinational Logic Circuits**. Transactions of the American Institute of Electrical Engineers, 1953.

KARZALIS S. A.; KALOMIROS, J.; KALAITZIS, V. **A. Cartesian Genetic Programming Approach for Evolving Optimal Digital Circuits**. Conference Paper. Department of Informatics Engineering, Technological Educational Institute of Central Macedonia, Serres, Greece. 2015.

KOZA, J. R. **Genetic Programming: On the Programming of Computers by Means of Natural Selection**. Cambridge Massachusetts: MIT Press, 1992.

LINDEN, R. **Algoritmos Genéticos**, 3ª edição. EditoraCiênciaModerna, 2012.

MATHWORKS. **MATLAB for Windows User's Guide. Version R2015b**. The Math Works, 1991.

METCALF, P. **Function To Generate Truth Table Condition Vectors**. MATLAB Central, 2011. Disponível em: <http://www.mathworks.com/matlabcentral/fileexchange/30008-function-to-generate-truth-table-condition-vectors>. Acesso em: abr. 2015.

MICHALEWICZ, Z. **Genetic Algorithms + Data Structures = Evolution Programs**. Artificial Intelligence Series. Springer-Verlag: Berlin, 1992.

MILLER, J. F. **Cartesian Genetic Programming**. Springer, 2011.

MITCHELL, M. **An Introduction to Genetic Algorithms**, MIT Press. 1999. Disponível em: <http://www.boente.eti.br/fuzzy/ebook-fuzzy-mitchell.pdf>. Acesso em: abr. 2015.

MOORE, E. F. **Gedanken-experiments on Sequential Machines.** Automata Studies, Princeton University Press, Princeton, N.J. 1956.

POLI, R. et al. **A Field Guide to Genetic Programming**, Lulu Enterprises, UK Ltd: United Kingdom, 2008.

RUOHONEN, K. **Graph Theory**. 2010. Disponível em: <http://math.tut.fi/~ruohonen/GT_English.pdf>. Acesso em: 2 ago. 2015.

SILVA, S. **GPLAB: A Genetic Programming Toolbox for MATLAB**. University of Coimbra, Portugal, 2007. Disponível em: <http://klobouk.fsv.cvut.cz/~leps/teaching/mmo/data/gplab.manual.3.pdf>. Acesso em: 10 ago. 2015.

SIVANANDAM, S. N.; DEEPA, S. N. **Introduction to Genetic Algorithms**. 1 ed. Springer, 2007.

SEKANINA L.; VASICEK Z. **Evolutionary Computing in Approximate Circuit Design and Optimization**. 1st Workshop On Approximate Computing, WAPCO 2015. Amsterdam, Holland. 2015.

SHANTI, A. P.; PARTHASARATHI R. **Evolution of Asynchronous Sequential Circuits**. Conference Paper. Anna University, India. 2005.

SOLEIMANI, P. et al. **Using Genetic Algorithm in the Evolutionary Design of Sequential Logic Circuits**. Ijcsi International Journal of Computer Science Issues**,** Tehran, Iran, v. 8, n. 5, 2011.

VERILOG. **Verilog-A Language Reference Manual**. 1 ed. Open Verilog International, 1996.

WALKER, J. A.; MILLER, J. F. **The Automatic Acquisition, Evolution and Reuse of Modules in Cartesian Genetic Programming**. IEEE Transactions On Evolutionary Computation, [s.l.], v. 12, n. 4, p. 397-417, ago. 2008. Institute of Electrical & Electronics Engineers (IEEE).

Apêndice A - Utilização do Algoritmo em MATLAB

A.1 Tutorial por Interface Gráfica

Para utilizar o programa desenvolvido, basta chamar a função *ae_gui* no console de comandos do MATLAB (Figura 33). O usuário informa no campo *Gerações* a quantidade máxima de gerações que o algoritmo deverá evoluir. No campo *Indivíduos* é necessário informar o tamanho da população a qual será utilizada e em *Tamanho da rede,* informa-se no primeiro campo a quantidade de linhas e no segundo a quantidade de colunas que esta possuirá. Além disso, a interface permite também a entrada da taxa de mutação (de 0 a 1) que será utilizada no operador de mutação. Também, é possível selecionar os arquivos de entrada e saída, bem como o tipo de circuito (combinacional ou sequencial) e a quantidade de estados da máquina (mesmo se for circuito combinacional, deve-se informar zero nesse campo). A janela abaixo é utilizada para mostrar os gráficos de execução (gerações passadas, avaliação do melhor indivíduo da geração, taxa de mutação e avaliação do pior indivíduo).

No menu *Primitivas* (Figura 34), escolhem-se quais primitivas lógicas serão usadas.

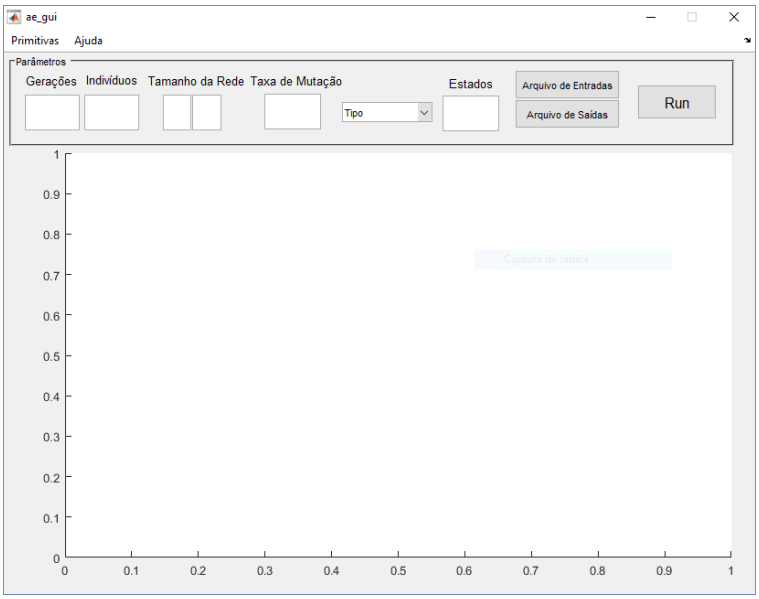

Figura 33: Janela principal da interface gráfica.

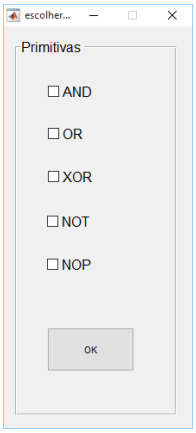

Figura 34: Menu de primitivas a serem escolhidas.

A.2 Tutorial completo por linha de comando

Para mais informações dos parâmetros passados para cada função, basta digitar:

help <nome da função>

Execução do AG por linha de comando:

1. Executar:

***algoritmoGenetico*(nger,ncrom,l,c,taxaMut,entradaTxt,saidaTxt,portas,tipo,qntEstados)**

A função *algoritmoGenetico* realiza a chamada de funções na seguinte ordem (cada uma com seus parâmetros adequados):

1.1. *geraPop*(ncrom,tam,c,l,usarPortas,n)*;*

1.2. *decodifica*(ncrom,c,l,s,nLinhas,pop,entradas,saidas,sgenes,sCalc,erro)*;*

1.3. *mutação*(ncrom,tam,c,l,usarPortas,taxaMut,n,pop,pai)*;*

1.4. *crossover2pontos*(tam,pais).

Depois do Algoritmo Genético terminar de evoluir e encontrar uma solução, é exibido na tela informações referentes ao processo de evolução e o fenótipo da solução de forma gráfica.

1.5. O fenótipo é desenhado com a função: *plotFen*(l,c,n,s,pai).

A função *plotFen* chama internamente as seguintes funções:

1.5.1. *destacaSolucao*(k,pai,l,caminho);

1.5.2. *plotCaminho*(aNo,lNo,aObj,lObj,lES,l,c,pai,caminho);

1.6. Além de gerar informações do processo de evolução e o fenótipo da solução, são também gerados automaticamente os arquivos *cromossomo.txt* e *parametros.txt*

- O arquivo *cromossomo.txt* contém o cromossomo solução codificado em hexadecimal para ser lido pela arquitetura flexível;
- O arquivo *parâmetros.txt* contém os parâmetros necessários para informar à arquitetura flexível.

2. Abrir o projeto *geraHardware* no Quartus II;

3. Modificar parâmetros informados em *parametros.txt* no cabeçalho da função *circuitoCompleto*;

Compilar o projeto.