UNIVERSIDADE FEDERAL DO TOCANTINS
CAMPUS UNIVERSITÁRIO DE PALMAS
CURSO DE CIÊNCIA DA COMPUTAÇÃO
TRABALHO DE CONCLUSÃO DE CURSO II

AVALIAÇÃO E TESTES DE HEURÍSTICAS PARA OTIMIZAÇÃO DE SISTEMAS DIGITAIS

Wandro Bequiman Maciel

Orientador: Me. Tiago da Silva Almeida

Palmas
Maio de 2017

AVALIAÇÃO E TESTES DE HEURÍSTICAS PARA OTIMIZAÇÃO DE SISTEMAS DIGITAIS

Wandro Bequiman Maciel

Trabalho de Conclusão de Curso II apresentado ao Curso de Ciência da Computação, CUP, da Universidade Federal do Tocantins, como parte dos requisitos necessários à obtenção do título de Bacharel em Ciência da Computação.

PALMAS, TO – BRASIL
MAIO DE 2017

Dados Internacionais de Catalogação na Publicação (CIP)
Sistema de Bibliotecas da Universidade Federal do Tocantins

M152a Maciel, Wandro Bequiman.

Avaliação e Testes de Heurísticas para Otimização de Sistemas Digitais. / Wandro Bequiman Maciel. – Palmas, TO, 2017.

74 f.

Monografia Graduação - Universidade Federal do Tocantins – Câmpus Universitário de Palmas - Curso de Ciências da Computação, 2017.

Orientador: Tiago Da Silva Almeida

1. Sistemas Digitais. 2. Algoritmo Genético. 3. Heurísticas. 4. Refinamento. I. Título

CDD 004

Elaborado pelo sistema de geração automatica de ficha catalográfica da UFT com os dados fornecidos pelo(a) autor(a).

Dedico este trabalho em memória de meu avô Boa Ventura Moreira Beckman. À meus pais, Maria de Lourdes Bequiman Maciel e Jaime Alves Maciel.

Resumo

Este trabalho foi elaborado para interagir através de um código intermediário, com um *framework* de síntese de sistemas eletrônicos em alto nível de abstração a fim de otimizar circuitos combinacionais. O *framework* pode ser classificado como uma ferramenta CAD (*Computer Aided Design*) que interpreta otimiza e traduz circuitos digitais representados em diagrama esquemático para a representação em linguagem de descrição de hardware. Este trabalho em específico atua exclusivamente na parte de refinamento onde utiliza-se de uma implementação de Algoritmo Genético tendo enfoque na minimização de circuitos considerando seu custo de produção. Para o desenvolvimento do Algoritmo foram utilizadas técnicas de implementação e casos de testes adotados de outros trabalhos como a função paridade ímpar e comparador. Os resultados foram comparados com o objetivo de avaliar a eficiência do Algoritmo também em relação aos resultados obtidos por dois métodos tradicionais da bibliografia que utilizam da álgebra booleana para a minimização, são eles, mapa de Karnaugh e Quine-McCluskey. O Algoritmo Genético foi eficaz na maioria dos casos de testes e destacou-se como principal desvantagem desta abordagem, o fraco desempenho temporal.

Palavra-chave: Sistemas Digitais. Algoritmo Genético. Heurísticas. Refinamento.

Abstract

This work was developed to interact through an intermediate code, with a framework of synthesis of electronic systems in high level of abstraction in order to optimize combinational circuits. The framework can be classified as a CAD (Computer Assisted Design) tool that interprets optimizes and translates digital circuits represented in schematic diagram for hardware description language representation. This work works exclusively in the part of refinement where a Genetic Algorithm implementation is used, focusing on the minimization of circuits considering its cost of production. For the development of the Algorithm, we used implementation techniques and test cases adopted from other works, such as parity and comparator function. The results were compared with the objective of evaluating the efficiency of the Algorithm also in relation to the results obtained by two traditional methods of the bibliography that use boolean algebra for minimization, which are Karnaugh and Quine-McCluskey maps. The Genetic Algorithm was effective in most of the test cases and the main disadvantage of this approach was the poor temporal performance.

Keywords: Digital Sytems. Genetic Algorithm. Heuristic. Refinement.

Lista de Figuras

2.1 Porta lógica *AND* 5
2.2 Porta lógica *OR* 5
2.3 Porta lógica *NOT* 6
2.4 Porta lógica *NOR*. 6
2.5 Porta lógica *NAND*. 7
2.6 Diagrama genérico de um circuito combinacional. 8
2.7 Representação geral de um circuito sequencial. 9
2.8 Estrutura estável realimentada independente de entradas. 9
2.9 Símbolo geral para um flip-flop e definição dos seus dois estados de saída possíveis. 10
2.10 Mapa de Karnaugh com os pares de 1s adjacentes. 12

3.1 Matriz bidimensional que mostra o arranjo das portas lógicas em níveis, onde cada porta do nível $j+1$ (para $j \geq 1$), pode receber suas entradas de qualquer uma das saídas do nível anterior. 19
3.2 Codificação utilizada para cada um dos elementos da matriz bidimensional que representa o circuito. 19
3.3 Formato de dados do cromossomo: A matriz de conexões e representação de circuito utilizando apenas portas NOR. 23
3.4 Circuito gerado pelo Synopsys Design Compiler com custo de 28 BCs. . . . 24
3.5 Circuito gerado pelo AG com custo de 24 BCs. 24
3.6 Fluxograma do sistema. 28
3.7 Exemplo 1: Estrutura do arquivo de entrada (matriz). 28
3.8 Exemplo 2: Estrutura do arquivo de entrada (matriz). 29

4.1 Processo do projeto de otimização 33
4.2 Exemplo de código intermediário JSON (oriundo da etapa de compilação) de entrada da etapa de otimização, a qual este trabalho implementa 34
4.3 Cromossomo formado a partir da matriz de conexões. 38
4.4 Circuito da matriz de conexões da Figura 4.1 representado com portas NOR. 38
4.5 Função de cruzamento 41
4.6 Função de mutação 41
4.7 Diagrama de classe do pacote Entities 43
4.8 Diagrama de classe para pacote Controller 44
4.9 Diagrama de classe do pacote Circuit JSON 45
4.10 Diagrama de classe para pacote Circuit Entities 45

5.1 Gráfico melhores de 5 execuções da função maioria. 49
5.2 Gráfico das cinco piores execuções da função maioria 50
5.3 Circuito minimizado por: (a) Karnaugh, McCluskey e (b) GA. 51
5.4 Gráfico das cinco execuções de melhores indivíduos da função paridade. . . 52
5.5 Gráfico das cinco execuções de piores indivíduos da função paridade. 52
5.6 Circuito minimizado por: (a) Karnaugh, McCluskey e (b) GA. 53
5.7 Gráfico das cinco execuções de melhores indivíduos da função comparador. 55

5.8 Gráfico das cinco execuções de piores indivíduos da função comparador . . 55
5.9 Diagrama do circuito resultante do mapa de karnaugh. 56
5.10 Diagrama do circuito resultante do GA. 57

Lista de Tabelas

2.1 Tabela verdade para a operação AND 5
2.2 Tabela verdade para a operação OR 5
2.3 Tabela verdade para operação NOT 6
2.4 Tabela verdade para a operação NOR. 6
2.5 Tabela verdade para operação $NAND$. 7
2.6 Mintermos e Maxtermos para três variáveis booleanas 11
2.7 Tabela verdade para a função de quatro variáveis do exemplo. 12
2.8 (a) mintermos separados em grupos pela quantidade de 1s (b) implicantes primos formados da relação G1 com G2. 14
2.9 Implicantes primos essenciais (m_3, m_8) em destaque entre os implicantes primos não essenciais (m_2, m_{10}). 14
2.10 Gráfico de implicantes primos essenciais para a função $\sum m(2, 3, 8, 10)$. . . 14

3.1 Tabela verdade para o exemplo apresentado. 21
3.2 Resultados para o exemplo produzidos pelo NGA. 21
3.3 Resultados para o exemplo produzidos pelo BGA. 22
3.4 Resultado para o exemplo produzido por um projetista humano. 22
3.5 Representação Matricial do Circuito. 25
3.6 Resultados obtidos pelo método proposto e método de Karnaugh. 27

5.1 Parâmetros utilizados nos testes do GA. 48
5.2 Custos de todos os testes em BCs. 48
5.3 Tabela verdade para a função Maioria. 48
5.4 Testes da função Maioria. 50
5.5 Tabela verdade da função paridade ímpar 51
5.6 Paridade ODD. 53
5.7 Tabela verdade da função comparador. 54
5.8 Resultado de dez execuções da função comparador. 56

Sumário

1 Introdução **1**
1.1 Justificativa . . . 2
1.2 Estrutura do Trabalho . . . 3

2 Fundamentação Teórica **4**
2.1 Álgebra Booleana . . . 4
2.1.1 Operações Básicas . . . 4
2.1.2 Universalidade das portas NAND e NOR . . . 6
2.1.3 Descrevendo circuitos através de expressões booleanas . . . 7
2.1.4 Circuitos Lógicos Combinacionais e Sequenciais . . . 7
2.1.5 Flip-Flops . . . 9
2.2 Representação de Funções Booleanas . . . 10
2.3 Simplificação de Circuitos Lógicos . . . 11
2.3.1 Mapas de Karnaugh . . . 11
2.3.2 Método de Quine-McCluskey . . . 13
2.4 Algoritmos Genéticos . . . 14
2.4.1 Operações Básicas . . . 15

3 Trabalhos Relacionados **18**
3.1 Evolução Automatizada de Projeto de Circuitos Combinacionais . . . 18
3.1.1 Representação do Cromossomo . . . 19
3.1.2 Função de Fitness . . . 20
3.1.3 Resultados . . . 20
3.2 Síntese de Redes Lógicas Multiníveis de Custo Mínimo Via Algoritmo Genético . . . 21
3.2.1 Representação do Cromossomo . . . 22
3.2.2 Função de Fitness . . . 23
3.2.3 Resultados . . . 24
3.3 EHW - Aplicado à Síntese de Circuitos Digitais Usando Representação por Portas Lógicas . . . 25
3.3.1 Representação do Cromossomo . . . 25
3.3.2 Função de Fitness . . . 26
3.3.3 Resultados . . . 26
3.4 Síntese de Circuitos Digitais Utilizando Computação Evolutiva . . . 27
3.4.1 Representação do Cromossomo . . . 28
3.4.2 Função do Fitness . . . 29
3.4.3 Resultados . . . 30

4 Metodologia **31**
4.1 Métodos utilizados . . . 31
4.2 Código intermediário estruturado em JSON . . . 32
4.3 O Algoritmo Genético . . . 33
4.3.1 População Inicial . . . 36
4.3.2 Matriz de conexões de portas NOR . . . 37

4.3.3 Cromossomo 37
4.3.4 Função do Fitness 38
4.3.5 Crossover e Mutação 40
4.3.6 Critério de Parada 41
4.4 Método de avaliação 42
4.5 Diagramas de Classe 42
4.5.1 Pacote *Entities* 42
4.5.2 Pacote *Controller* 44
4.5.3 Pacote *Circuit JSON* 45
4.5.4 Pacote *Circuit Entities* 45

5 Resultados **47**
5.1 Casos de Testes 47
5.1.1 Função Maioria 48
5.1.2 Circuito de Paridade ODD 51
5.1.3 Comparador de Magnitude 54
5.2 Considerações finais sobre o capítulo 57

6 Conclusões **58**

Referências Bibliográficas **60**

1 Introdução

As fases de concepção, análise, implementação e testes de sistemas eletrônicos são importantes para o desenvolvimento de tecnologias de qualidade. Um projeto de *hardware* ou *software* elaborado de forma coerente permite a redução significativa de tempo e custo monetário.

A sistematização das etapas de projeto nas fases de concepção, análise, implementação e testes é de vital importância para a redução dos custos associados. Nesse sentido alguns trabalhos representaram um marco na sistematização de projetos de sistemas eletrônicos e de *software*. Em projeto de sistemas eletrônicos é importante citar o trabalho de [1]. Dadas as diferentes representações de um determinado sistema eletrônico, o projeto pode ser dividido em três níveis denominados de: comportamental, estrutural e físico. Gajski e Kuhn [1] sistematizaram a representação de sistemas eletrônicos de acordo com o contexto da época. Assim, Riesgo e Torre [2] atualizaram as etapas de projeto para as novas representações que surgiram ao longo dos anos.

A representação em altos níveis de abstração dificulta a implementação e fabricação de dispositivos eletrônicos. Por isso, deu-se origem a metodologia conhecida como *top-down*, [1] [2]. As etapas dessa metodologia são conhecidas como "processo de síntese", onde a partir de um determinado nível de abstração é obtido um nível mais baixo de abstração com a representação equivalente. O processo de síntese também se aplica à mudança de representação, ou seja, do comportamental para o estrutural ou do estrutural para o físico.

Mesmo com uma metodologia de elaboração e análise de projetos bem definida e atualizada [2], eventualmente, outras formas de representação surgiram para tentar automatizar os projetos de sistemas eletrônicos. Assim, Gerstlauer e Haubelt [3] propuseram uma generalização entre as formas de representação e síntese de sistemas eletrônicos com a adição de novas características. Essa adição é a representação em *software*, porém ainda existe a representação em *hardware*. A representação em *software* é mais evidente em sistemas embarcados ou sistemas de aplicação específica, pois atualmente existem muitas ferramentas computacionais proprietárias para automação de projetos. Dessa forma, não é necessária a construção do *hardware*, já que muitas vezes ele é reprogramável.

O nível ESL (*Electronic System Level*) é um modelo comportamental que muitas vezes

representa um tipo de rede de processos de comunicação. O modelo ESL é tipicamente um modelo estrutural constituído pelos componentes de arquitetura. A tarefa de síntese ESL é o processo de seleção de uma arquitetura de plataforma apropriada, determinando o mapeamento do modelo comportamental em uma arquitetura, e gerando a correspondente implementação do comportamento executado na plataforma. Se selecionado, os componentes desse modelo refinado são então usados como entrada para as etapas de projeto em níveis mais baixos de abstração [3].

Com base nas definições da literatura [1, 2, 3], o *framework* se baseia na metodologia *top-down* e de projetos em nível ESL, pois ambos são amplamente empregado em projetos de pesquisa atuais [4, 5, 6, 7, 8]. O objetivo não é somente o desenvolvimento de um *software* de síntese, mas um conjunto de componentes de *software* que possibilitem a reutilização dos componentes em outros projetos com uma arquitetura padronizada. Assim, o presente trabalho visa o desenvolvimento de um componente de otimização de circuitos para auxiliar, complementar e interagir com o *framework* de síntese de sistemas eletrônicos.

1.1 Justificativa

Atualmente a necessidade de automação de projetos de sistemas digitais tem sido grande devido a busca por maior eficiência, mais agilidade, baixo custo e maior qualidade. A tarefa de se projetar circuitos digitais de grande complexidade pode se tornar inviável sem a utilização de ferramentas e metodologias que auxiliam nesse processo.

Segundo Almeida *et.al* [9] as ferramentas computacionais são indispensáveis hoje em razão do aumento da complexidade dos projetos e a necessidade de gerir grande quantidade de dados relacionados. Com o desenvolvimento de novas metodologias e ferramentas torna-se necessária, em especial, a criação de ferramentas de projeto auxiliado por computador - CAD *(Computer Aided Design)*.

As ferramentas CAD podem ser melhores compreendidas como sistemas de gerenciamento de informações de projeto, juntamente com a criação de gráficos e simulação dos projetos criados. Essas simulações podem ser usadas, compartilhadas, publicadas, republicadas e reutilizadas em diferentes formatos, escalas e níveis de detalhe [10].

Com base nesta abordagem destaca-se a importância desta pesquisa da construção

deste projeto para automação e auxílio na tarefa de otimização e refinamento de circuitos lógicos utilizando a metodologia de síntese de modelos *top-down* descrita por Riesgo e Torre [2].

O objetivo principal desse trabalho foi a utilização de um Algoritmo Genético para otimizar os modelos de circuitos com custo mínimo. Os resultados foram comparados com os resultados obtidos pelo método de Quine-McCluskey e mapa de Karnaugh. Foram testadas otimizações para três modelos de circuitos: Função Maioria, Circuito Paridade Ímpar e Comparador de Magnitude.

1.2 Estrutura do Trabalho

O Capítulo 2 apresenta a fundamentação teórica utilizada neste trabalho como conceitos básicos e resumidos de álgebra booleana, métodos de minimização de expressões booleanas e conceitos de Algoritmos Genéticos. Posteriormente, no Capítulo 3, estão apresentados alguns trabalhos relacionado à esta pesquisa, com ênfase na modelagem e codificação utilizada do Algoritmo Genético e seus respectivos resultados. No Capítulo 4 é apresentada a metodologia proposta deste trabalho e uma explanação sobre os métodos utilizados e como foram implementados. O Capítulo 5 apresenta os resultados dos testes realizados e comparações entre os métodos utilizados. Por fim, no Capítulo 6, são descritas as conclusões sobre o trabalho.

2 Fundamentação Teórica

Neste capítulo são apresentados conceitos básicos utilizados neste trabalho. Inicialmente são mostrados os fundamentos da álgebra de Boole e métodos de minimização de expressões lógicas como Mapas de Karnaugh e Quine-McCluskey. Por fim, é apresentada uma revisão sobre a base teórica de Algoritmos Genéticos.

2.1 Álgebra Booleana

Proposta por George Boole, a área hoje denominada Álgebra Booleana trata de problemas relacionados à lógica binária. Essa lógica também forma a base para computação em sistemas modernos. Qualquer algoritmo ou circuito eletrônico digital pode ser representado por um sistema de equações booleanas [11].

Da mesma forma que são usados símbolos como x e y para representar valores numéricos desconhecidos na álgebra comum, a álgebra booleana usa símbolos para representar uma expressão lógica que possui um de dois valores possíveis: verdadeiro ou falso. A principal diferença entre a álgebra booleana e a álgebra convencional é que, na álgebra booleana, as constantes e variáveis podem ter apenas dois valores possíveis, 0 ou 1 [12]. Operações sobre variáveis são realizadas utilizando operadores unários, que possuem uma variável como entrada, e binário, que possuem duas [13]. A seção 2.1.1 tratará das operações básicas.

2.1.1 Operações Básicas

As funções AND, OR e NOT são as três funções básicas da álgebra booleana. Todas as outras funções podem ser representadas em termos destas três operações.

Operação AND e Porta AND

Por definição, o resultado do AND será '1' se ambas as entradas forem '1'. Para todos os outros casos a saída será '0'. Também chamada de multiplicação, o símbolo usualmente utilizado é o '.', assim como na multiplicação da aritmética.

A porta AND pode ter duas ou mais entradas e sua saída é igual à combinação das entradas por meio da operação AND. A Figura 2.1 mostra um exemplo de porta lógica de duas entradas e seu comportamento ilustrado na Tabela 2.1.

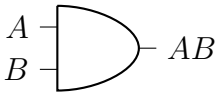


Figura 2.1: *Porta lógica AND*

Tabela 2.1: *Tabela verdade para a operação AND*

A	B	AB
0	0	0
0	1	0
1	0	0
1	1	1

Operação OR e Porta OR

Também denominada adição lógica, o OR resulta '1' se pelo menos uma das variáveis de entrada vale '1', nos demais casos o resultado será '0' obviamente. O símbolo mais usual para representar a operação OR é o '+' tal como na adição algébrica.

A porta OR, por sua vez, é um circuito lógico que pode ter duas ou mais entradas e sua saída é a combinação resultante da operação OR. Veja a representação do seu símbolo lógico na Figura 2.2 e tabela verdade na Tabela 2.2.

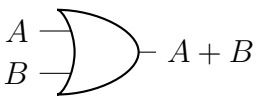


Figura 2.2: *Porta lógica OR*

Tabela 2.2: *Tabela verdade para a operação OR*

A	B	A+B
0	0	0
0	1	1
1	0	1
1	1	1

Operação NOT (INVERSOR)

A operação NOT, também denominada de complementação, é a operação cujo resultado é simplesmente o valor complementar ao que a variável apresenta. Por exemplo, se a entrada for '1' a saída será '0', ou seja, uma simples inversão do valor de entrada. Uma variável denominada A tem sua complementação representada como $\overline{A}$. A Figura 2.3 mostra o símbolo para o circuito NOT e a tabela verdade na Tabela 2.3.

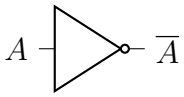


Figura 2.3: *Porta lógica NOT*

Tabela 2.3: *Tabela verdade para operação NOT*

A	$\overline{A}$
0	1
1	0

2.1.2 Universalidade das portas NAND e NOR

Outros dois tipos de portas lógicas bastante utilizadas em circuitos digitais são as portas NAND e NOR. Essas portas combinam as três operações básicas AND, OR e NOT de modo que a escrita das expressões booleanas sejam mais simples, principalmente quando se trata de uma representação em símbolos lógicos. As Figuras 2.4 e 2.5 apresentam um exemplo de porta lógica NOT e NAND, respectivamente, e suas respectivas tabelas verdade estão representadas nas Tabelas 2.4 e 2.5.

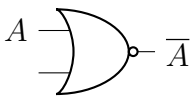


Figura 2.4: *Porta lógica NOR.*

Tabela 2.4: *Tabela verdade para a operação NOR.*

A	**B**	$\overline{A+B}$
0	0	1
0	1	0
1	0	0
1	1	0

Todas as expressões booleanas consistem em várias combinações das operações básicas OR, AND e INVERSOR. Portanto, qualquer expressão pode ser implementada usando combinações de portas NAND e NOR. Isso porque as portas NAND, em combinações apropriadas, podem ser usadas para implementar cada uma das operações booleanas OR, AND e INVERSOR [12].

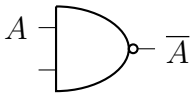


Figura 2.5: *Porta lógica NAND.*

Tabela 2.5: *Tabela verdade para operação $NAND$.*

A	**B**	$\overline{AB}$
0	0	1
0	1	1
1	0	1
1	1	0

2.1.3 Descrevendo circuitos através de expressões booleanas

Uma expressão booleana sempre produz um valor binário, uma vez que é composta de uma combinação de constantes e variáveis booleanas (verdadeiro ou falso) e conectivos lógicos. Cada expressão booleana representa a função de um circuito.

Uma característica favorável das expressões booleanas e que o valor resultante está contido no conjunto binário {0,1} e, como consequência, pode representar essas funções a partir de tabelas verdade [13]. Uma das vantagens da utilização de tabelas verdade seria a possibilidade de análise de uma porta ou combinação lógica de cada vez, permitindo que se confira facilmente o trabalho [12]. Quando a operação de um circuito é definida por uma expressão booleana, é possível desenhar o diagrama de circuito lógico a partir desta expressão [12].

2.1.4 Circuitos Lógicos Combinacionais e Sequenciais

Circuitos lógicos são classificados em dois tipos: combinacionais e sequenciais. Os circuitos combinacionais são aqueles nos quais as saídas são determinadas em função apenas das entradas atuais. Os circuitos sequenciais, são aqueles nos quais as saídas dependem não apenas das entradas atuais, mas também de dados prévios nos instantes anteriores.

Lógica Combinacional

Um circuito combinacional é constituído por um conjunto de portas lógicas, as quais determinam os valores das saídas diretamente a partir dos valores atuais das entradas e realiza uma operação de processamento de informação, a qual pode ser especificada por meio de um conjunto de equações booleanas. No caso, cada combinação de valores de entrada pode ser vista como uma informação diferente e cada conjunto de valores de saída representa o resultado da operação.

A Figura 2.6 ilustra a representação geral de um circuito combinacional com N entradas as quais passam por uma lógica combinacional até obter N saídas [14].

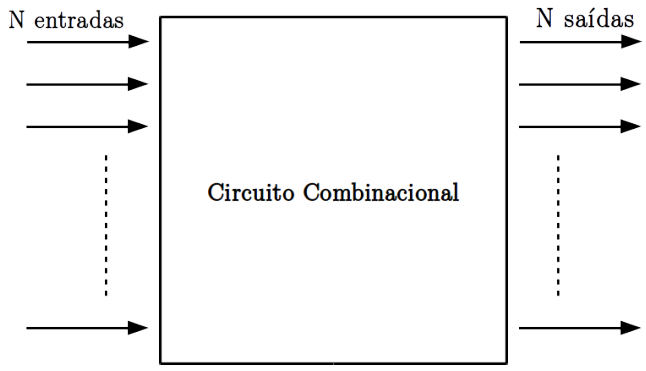


Figura 2.6: *Diagrama genérico de um circuito combinacional.*

Lógica Sequencial

Circuitos sequenciais caracterizam-se por apresentar saídas que dependem não só dos valores atuais das entradas, mas também da sequência com que os valores são aplicados nas entradas. São constituídos por uma lógica combinacional e também por células de memória que armazenam o estado atual do sistema. O estado atual do sistema define, em conjunto com as entradas, o comportamento futuro das saídas e dos próximos estados. A Figura 2.7 ilustra uma representação geral de um circuito sequencial, com portas de entradas e saídas, uma lógica combinacional e células de memória que podem definir o comportamento do circuito [14].

São exemplos de circuitos sequenciais: *Flip-Flops*, *Latches*, Registradores, Contadores, Máquinas de Estados Finitos entre outros.

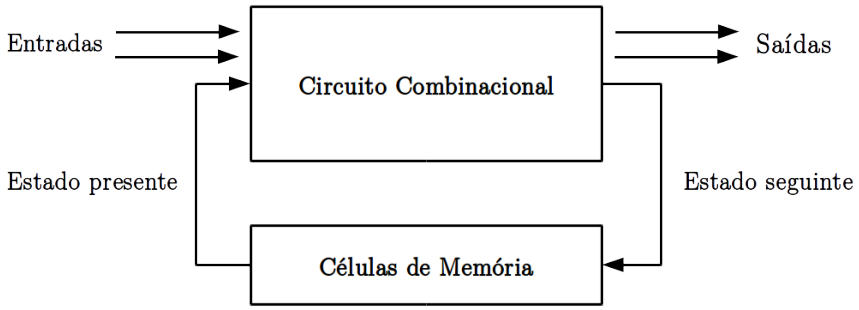


Figura 2.7: *Representação geral de um circuito sequencial.*

2.1.5 Flip-Flops

De acordo com a definição de circuitos sequenciais descrita no início da seção 2.1.4, em relação à memória e realimentação, o *flip-flop* é o elemento de memória mais importante, o qual é implementado a partir de portas lógicas. Embora uma porta lógica, por si só, não tenha capacidade de armazenamento, algumas delas podem ser conectadas entre si de tal forma que permita o armazenamento de informação. Algumas formas diferentes de arranjo de portas são usadas para produzir *flip-flops* [12]. *Flip-Flop* é um elemento capaz de guardar o valor de 1 bit [15]. A Figura 2.8 mostra uma estrutura realimentada independente de entrada.

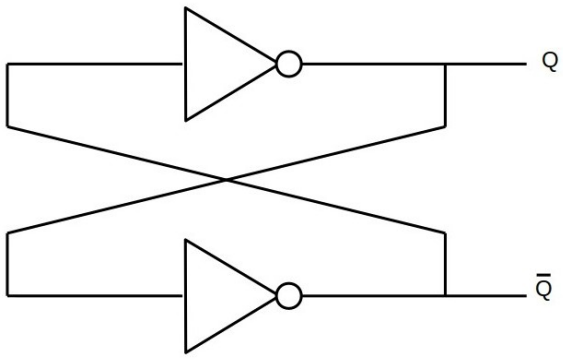


Figura 2.8: *Estrutura estável realimentada independente de entradas.*

A Figura 2.9 mostra o símbolo geral para um *flip-flop*. A saída Q é denominada saída *normal* e $\overline{Q}$ a saída invertida. Quando (Q = 1) o estado de saída é chamado de ALTO, 1 ou de *SET*, já quando (Q = 0) é chamado de estado BAIXO, 0 ou *RESET*.

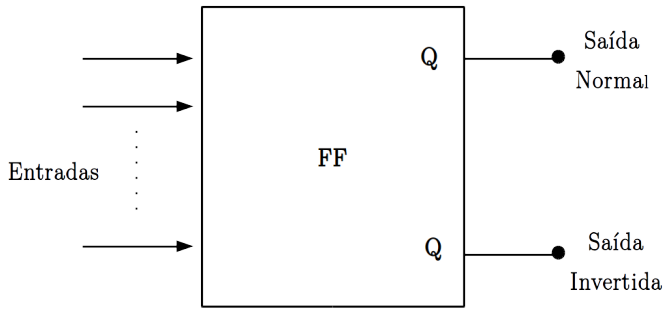


Figura 2.9: *Símbolo geral para um flip-flop e definição dos seus dois estados de saída possíveis.*

2.2 Representação de Funções Booleanas

A criação de funções booleanas mostra ser bastante flexível e diversificada. Porém nem todas são únicas. É possível que duas funções distintas produzam a mesma saída, como por exemplo $f(A, B) = A \cdot B$ e $g(A, B) = \overline{\overline{A} + \overline{B}}$. Neste caso, há uma infinidade de funções booleanas para uma dada tabela verdade. Torna-se necessário uma padronização visando apenas uma função booleana possível para cada configuração da tabela verdade [13].

Uma padronização de funções booleanas pode ser alcançada através da forma canônica de expressões lógicas. Há duas formas principais para expressões canônicas, conhecidas como a forma soma de produtos (mintermos m_n) e a forma produto de somas (maxtermos M_n) [16]. Em cada termo soma e em cada termo produto, todas as variáveis da função estão presentes. As formas descritas são chamadas canônicas devido a estas características.

Uma função booleana pode ser expressa algebricamente a partir de uma tabela verdade formando um *mintermo* para cada combinação das variáveis que produz '1' na função. Em seguida, acrescenta-se o operador '+' unindo todos os termos [17].

Para um circuito com entradas $A_1, A_2, ..., A_n$, um *mintermo* é um produto em que cada variável de entrada ou seu complemento aparecem apenas uma vez. Um *maxtermo* é o oposto, ou seja, apresenta a soma de todas as variáveis de entrada que aparecem uma única vez [18].

A Tabela 2.6 lista os termos associados à cada combinação de entradas para uma função booleana de três variáveis. Por exemplo, a função da tabela é determinada pelas combinações 001, 100, e 111 quando $f = 1$ sendo $f(A, B, C) = \overline{A}\overline{B}C + A\overline{B}\overline{C} + ABC = m_1 + m_2 + m_3$, a expressão lógica formada pelos *mintermos* descritos.

Tabela 2.6: *Mintermos e Maxtermos para três variáveis booleanas*

				Mintermos		Maxtermos	
A	B	C	Função	Termo	Designação	Termo	Designação
0	0	0	0	$\overline{A}, \overline{B}, \overline{C}$	m_0	$\overline{A}, \overline{B}, \overline{C}$	M_0
0	0	1	1	$\overline{A}, \overline{B}, C$	m_1	$\overline{A}, \overline{B}, C$	M_1
0	1	0	0	$\overline{A}, B, \overline{C}$	m_2	$\overline{A}, B, \overline{C}$	M_2
0	1	1	0	$\overline{A}, B, C$	m_3	$\overline{A}, B, C$	M_3
1	0	0	1	$A, \overline{B}, \overline{C}$	m_4	$A, \overline{B}, \overline{C}$	M_4
1	0	1	0	$A, \overline{B}, C$	m_5	$A, \overline{B}, C$	M_5
1	1	0	0	$A, B, \overline{C}$	m_6	$A, B, \overline{C}$	M_6
1	1	1	1	A, B, C	m_7	A, B, C	M_7

2.3 Simplificação de Circuitos Lógicos

Uma vez obtida a expressão de um circuito lógico, pode-se reduzir a uma forma mais simples que contenha um menor número de termos ou variáveis em um ou mais termos da expressão. Essa nova expressão pode, então, ser usada na implementação de um circuito equivalente ao circuito original, mas que contém menos portas lógicas e conexões [12]. Nas seções subsequentes, serão apresentados dois métodos para simplificação de circuitos: mapa de Karnaugh e Quine-McCluskey.

2.3.1 Mapas de Karnaugh

O mapa de Karnaugh é um método gráfico usado para simplificar uma função lógica ou para converter uma tabela verdade no seu circuito lógico correspondente, de uma forma simples e metódica. Embora um mapa de Karnaugh possa ser usado em problemas que envolvem qualquer número de variáveis de entrada, sua utilidade prática está limitada a cinco ou seis variáveis [12].

A principal desvantagem de se usar mapas de Karnaugh é que o algoritmo funciona bem para até seis variáveis de entrada, tornando-se impraticável para valores maiores. O número excessivo de células dificulta uma seleção razoável de saídas adjacentes [17].

O objetivo do mapa de Karnaugh é deixar explícito características redundantes de um circuito de forma que seja possível retirar termos desnecessários da expressão lógica. No exemplo a seguir, o mapa é construído a partir da tabela verdade da Tabela 2.7, de acordo com os *mintermos* que formam a função $f(A, B, C, D) = \overline{AB}C\overline{D} + \overline{AB}CD + A\overline{BCD} + A\overline{B}C\overline{D}$.

Tabela 2.7: *Tabela verdade para a função de quatro variáveis do exemplo.*

A	B	C	D	X	Mintermos
0	0	0	0	0	
0	0	0	1	0	
0	0	1	0	1	$\overline{A}\overline{B}C\overline{D}$
0	0	1	1	1	$\overline{A}\overline{B}CD$
0	1	0	0	0	
0	1	0	1	0	
0	1	1	0	0	
0	1	1	1	0	
1	0	0	0	1	$A\overline{B}\overline{C}\overline{D}$
1	0	0	1	0	
1	0	1	0	1	$A\overline{B}C\overline{D}$
1	0	1	1	0	
1	1	0	0	0	
1	1	0	1	0	
1	1	1	0	0	
1	1	1	1	0	

Observa-se que na Figura 2.10 o mapa foi preenchido pelos '1's correspondentes da tabela verdade e agrupados em dois quadros (pares) formando dois grupos, sendo os '1's da linha superior horizontalmente adjacentes e os '1's da linha inferior também adjacentes, uma vez que, em um mapa de Karnaugh, a coluna mais à esquerda e a coluna mais à direita são consideradas adjacentes. Quando os pares de '1's superiores são agrupados, a variável D é eliminada, pois nos *mintermos* do grupo, $\overline{A}\overline{B}CD + \overline{A}\overline{B}C\overline{D}$, a variável em questão sofre modificação de valor, tornando-a nula. O mesmo acontece com a variável C do agrupamento inferior, resultando na função reduzida: $\overline{A}\overline{B}C + A\overline{B}\overline{D}$.

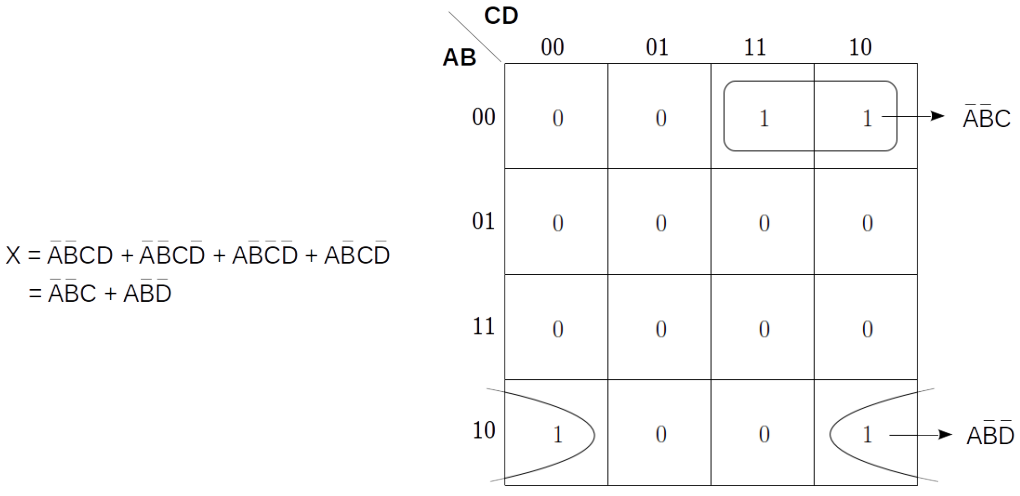


Figura 2.10: *Mapa de Karnaugh com os pares de 1s adjacentes.*

2.3.2 Método de Quine-McCluskey

O Método de Quine-McCluskey é tabular e supera a limitação associada ao mapa de Karnaugh. Um pouco tedioso para a computação manual, seu objetivo é fornecer um procedimento algorítmico para alcançar os implicantes primos de uma função lógica [19, 20]. Implicantes primos são todos os termos candidatos à inclusão na função simplificada, ou seja, os termos que são relevantes na formação da função reduzida equivalente [21].

Segundo [21], a redução de uma expressão booleana por um método de tabulação, envolve duas importantes atividades:

- Determinação dos implicantes primos;
- Seleção dos implicantes primos essenciais da função.

Para determinar quais termos são os implicantes primos é necessário comparar cada *mintermo* com todos os outros. Se eles diferem em apenas uma variável, ela é removida, formando assim um novo termo. Esse processo é repetido até que não seja mais possível combinar nenhum termo [13]. O exemplo a seguir detalha com mais clareza o processo de minimização de expressões lógicas que o método Quine-McCluskey determina.

Os *mintermos* utilizados são os mesmos apresentados na Tabela 2.7 da seção 2.3.1 para o mapa de Karnaugh. A equação 2.1 representa os mintermos da função do exemplo apresentado.

$$\sum m(2, 3, 8, 10) \tag{2.1}$$

Esses *mintermos* são divididos em grupos de acordo com a quantidade de 1s que possuem. Para este exemplo pode-se observar na Tabela 2.8 (a) que os grupos G1 e G2 separam os *mintermos* de apenas um dígito 1 dos que possuem dois. Os implicantes primos encontrados na comparação dos grupos G1 e G2 são mostrados na Tabela 2.8 (b). Nota-se que as lacunas são resultantes da comparação dos bits que se divergem, eliminando assim, a variável correspondente.

A Tabela 2.9 mostra em destaque os implicantes primos essenciais extraídos da comparação dos termos adjacentes que se repetem dos que não se repetem. Por exemplo, a linha correspondente ao termo (m_2, m_{10}) composto por mintermos redundantes, pode

ser eliminada. Os implicantes primos essenciais da Tabela 2.10 apresenta o último passo para este exemplo, onde não há nenhuma coluna com mais de uma combinação. Por fim, as expressões são somadas. Nesse caso, a função resultante foi $\overline{A}BC + A\overline{B}D$, a mesma encontrada pelo método do mapa de Karnaugh.

Tabela 2.8: *(a) mintermos separados em grupos pela quantidade de 1s (b) implicantes primos formados da relação G1 com G2.*

Grupo	Quantidade de 1s	mintermo	
G1	1	m_2	0010
		m_8	1000
G2	2	m_3	0011
		m_{10}	1010

(a)

Implicantes primos	
(m_2, m_3)	0 0 1 _
(m_2, m_{10})	_ 0 1 0
(m_8, m_{10})	1 0 _ 0

(b)

Tabela 2.9: *Implicantes primos essenciais (m_3, m_8) em destaque entre os implicantes primos não essenciais (m_2, m_{10}).*

	m_2	m_3	m_8	m_{10}	Implicantes Primos	Expressão
(m_2, m_3)	X	**X**			0 0 1 _	$\overline{A}BC$
(m_2, m_{10})	X			X	_ 0 1 0	$\overline{B}CD$
(m_8, m_{10})			**X**	X	1 0 _ 0	$A\overline{B}\overline{D}$

Tabela 2.10: *Gráfico de implicantes primos essenciais para a função $\sum m(2, 3, 8, 10)$.*

	m_2	m_3	m_8	m_{10}	Implicantes Primos Essenciais	Expressão
(m_2, m_3)	X	X			0 0 1 _	$\overline{A}\overline{B}C$
(m_8, m_{10})			X	X	1 0 _ 0	$A\overline{B}\overline{D}$

O algoritmo de Quine-McCluskey também possui limitações assim como o mapa de Karnaugh. Esse método tem seus limites práticos porque é NP-completo. Em outras palavras, o tempo de execução do Quine-McCluskey cresce exponencialmente com o tamanho da entrada [21].

2.4 Algoritmos Genéticos

Inspirados na teoria evolucionária de Darwin, seleção natural e nos processos biológicos envolvidos, os Algoritmos Genéticos (AGs) são algoritmos meta-heurísticos que representam uma classe de algoritmos de otimização os quais empregam mecanismo de pesquisa probabilísticos de soluções [22]. Apesar de envolver procedimentos aleatórios, os AGs se diferenciam dos demais, segundo [23], em quatro aspectos fundamentais por:

- trabalhar com codificação de parâmetros, ao invés dos padrões originais do problema;
- pesquisar soluções ótimas a partir de um conjunto de soluções e não apenas de uma;
- empregar uma função de avaliação para as diferentes soluções pesquisadas, codificadas em sequencias de comprimentos conhecidos como *strings*.
- utilizar regras probabilísticas e não probabilísticas, na pesquisa de novas soluções.

2.4.1 Operações Básicas

Nesta seção serão apresentadas os principais operadores utilizados em Algoritmos Genéticos.

Cromossomo

Um cromossomo é um conjunto de genes de uma espécie específica. Cada gene corresponde à uma característica particular e cada uma dessas características devem ser independentes para evitar a interação entre os genes. De uma forma ideal, não deve existir correlação entre os valores dos genes. O papel de um gene é dar características a um indivíduo.

Função de *Fitness*

A função de *fitness* ou função-objetivo, associa a cada cromossomo um valor correspondente à sua aptidão. A função de *fitness* não deve apenas indicar que um cromossomo é bom, mas também o quanto ele está próximo do ótimo. No entanto, o Algoritmo Genético irá se concentrar em soluções ótimas ou quase ótimas.

População

A população é um conjunto de n indivíduos (cromossomos) onde cada um deles representam uma possível solução para o problema em questão. A primeira população deve ter um número suficiente de indivíduos para obter resultados satisfatórios. Quanto maior for a população, mais fácil será para explorar o espaço de pesquisa.

Seleção

Esta etapa é executada logo após o cálculo da aptidão dos indivíduos. Sua implementação se baseia no processo de seleção natural, onde os indivíduos mais capazes possuem maior

probabilidade de gerar mais descendentes, enquanto que os menos capazes poderão ainda gerar descendentes, mas nesse caso em uma escala menor. A seguir apresenta-se três métodos de seleção mais utilizados:

1. **Roleta**: nesta técnica, a seleção dos cromossomos ocorre de forma proporcional ao valor de aptidão do indivíduo. Cromossomos que possuem maior aptidão ocuparão uma maior fração na roleta, enquanto que os cromossomos com menor aptidão ocupam obviamente um menor espaço, tendo uma menor probabilidade de serem escolhidos.

2. **Torneio**: este método se dá a partir da escolha de n cromossomos da população atual, de forma aleatória. Dentre os cromossomos escolhidos, aquele com maior *fitness* é selecionado para compor uma população intermediária. Posteriormente, os demais indivíduos são recolocados na população e realiza-se novamente o mesmo processo até que a população intermediária esteja completa.

3. **Estratégia Elitista**: A estratégia elitista consiste na troca de gerações de forma que, seleciona-se k indivíduos ($k \geq 1$), os quais possuem o melhor **fitness** para compor uma próxima geração, de modo que seus cromossomos serão preservados nas gerações subsequentes. Com isso, garante-se uma evolução da população.

Reprodução ou *Crossover*

Esta operação busca imitar o fenômeno da combinação genética. É nesta fase onde ocorrem a troca de segmentos (genes) entre pares de cromossomos selecionados na etapa de seleção anteriormente descrita. O objetivo é originar os novos indivíduos que virão a formar a população da geração seguinte. A ideia principal desta operação é propagar as características positivas dos indivíduos mais aptos da população garantindo que populações futuras possuam cromossomos de *fitness* muito melhores.

As formas mais comuns de troca de segmentos em AGs são as de ponto único (demostrada a seguir) e ponto duplo.

- Ponto único: é escolhido um ponto de corte aleatório em cada um dos cromossomos do par e a partir desse ponto separa-se o segmento para ser adicionado ao novo indivíduo (filho) realizando assim a troca de material genético e dando origem ao novo indivíduo.

Mutação

A operação de mutação é realizada após o processo de cruzamento e seu objetivo é modificar, de maneira aleatória, determinadas propriedades genéticas de uma população, garantindo a manutenção da diversidade genética. Um exemplo de mutação comumente utilizado é a mutação por troca, onde n pares de genes são sorteados e suas posições trocadas.

3 Trabalhos Relacionados

Neste capítulo são apresentados os trabalhos correlatos com foco nos modelos de codificação de cromossomos utilizados e outras operações genéticas consideradas relevantes. As seções estão organizadas em: representação do cromossomo, função de aptidão e resultados obtidos.

3.1 Evolução Automatizada de Projeto de Circuitos Combinacionais

O trabalho de Coello et.*al*[24] propõe uma metodologia baseada em AG para automatizar a concepção de circuitos lógicos combinatórios nos quais pretende minimizar o número total de portas lógicas utilizadas. Os resultados foram comparados com os produzidos por projetistas humanos e por uma outra abordagem baseada em AG. Coello et.*al* também analisa a importância de se usar a representação não-binária, apesar desta notação ser comum universalmente e utilizada em todos os tipos baseados em AG. Destaca também que, apesar de projetistas de circuitos tenderem a usar mais portas AND, OR ou NOT, a medida geral de otimização adotada no trabalho é o número total de portas utilizadas, independente da sua espécie.

O problema de interesse dos autores consiste em projetar um circuito que executa uma função desejada (especificada por uma tabela verdade), dado um determinado conjunto de portas lógicas disponíveis. No projeto de circuito usa-se vários critérios para expressões de custo mínimo.

A medida geral do custo de um circuito de acordo com Coello et.*al* [24], é o número total de portas utilizadas, independentemente de seu tipo. Esta medida é também proporcional ao custo total da peça do circuito.

Um grande número de portas presentes em um circuito pode fazer com que ele sofra uma pequena penalidade em sua velocidade.

3.1.1 Representação do Cromossomo

O modelo de representação do cromossomo escolhido pelos autores foi uma matriz bidimensional sugerida por Louis e Rawlins [25], em que cada elemento da matriz representa uma porta lógica, AND, NOT, OR, XOR e WIRE (fio), que recebe suas duas entradas de qualquer porta na coluna anterior como mostra a Figura 3.1.

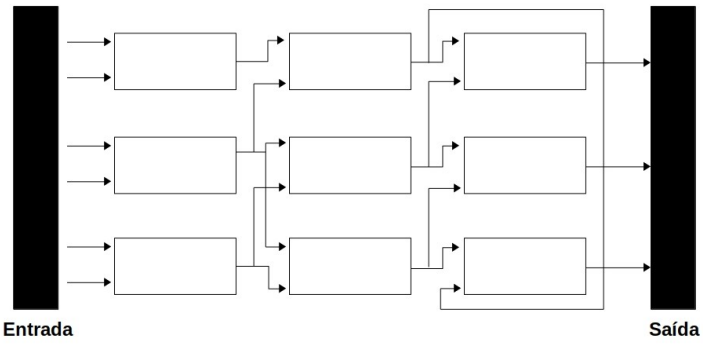


Figura 3.1: *Matriz bidimensional que mostra o arranjo das portas lógicas em níveis, onde cada porta do nível $j+1$ (para $j \geq 1$), pode receber suas entradas de qualquer uma das saídas do nível anterior.*

De maneira formal, qualquer circuito pode ser representado como uma matriz bidimensional de portas $S_{i,j}$, onde j indica o nível de uma porta, de modo que as portas mais próximas das entradas possuam valores menores para j (os valores de níveis nesse caso, são incrementados da esquerda para a direita na Figura 3.1). Para o j fixo, o índice i varia em relação às portas que estão "próximas" umas das outras no circuito, mas sem estar necessariamente ligadas.

Uma sequência cromossômica codifica a matriz apresentada na Figura 3.1 usando *Triplets* (trigêmeos) nos quais os dois primeiros elementos se referem a cada uma das entradas e o terceiro é a porta correspondente, como mostrado na Figura 3.2. Por fim, os autores argumentam que o modelo utilizado verifica se em alguns domínios, tais como otimização, alfabetos de maior cardinalidade provaram fornecer um melhor resultado em um período de tempo mais curto do que suas contrapartidas binárias [26].

Entrada 1	Entrada 2	Tipo de Porta

Figura 3.2: *Codificação utilizada para cada um dos elementos da matriz bidimensional que representa o circuito.*

3.1.2 Função de Fitness

A função é ligeiramente uma variação da sugerida por Louis e Rawlins [25], a qual consiste no número de respostas corretas obtidas pelo AG em relação à tabela verdade fornecida pelo utilizador. O fitness de um indivíduo x é dado por:

$$fitness(x) = \begin{cases} \sum_{j}^{p} = 1 f_j(x) & \text{se } x \text{ não factível} \\ \sum_{j}^{p} = 1 f_j(x) + w(x) & \text{caso contrário} \end{cases} \tag{3.1}$$

Nesse caso, p é o número de entradas da tabela verdade e, normalmente, p é igual a n^2, sendo n o número de entradas da tabela verdade, mas p pode também ser atribuído diretamente a um certo valor caso a tabela verdade tiver *"don't cares"*. O valor de $f_j(x)$ depende dos resultados produzidos pelo circuito (x) codificado pelo AG ($f_j(x)$ é inicializado em zero e, sempre que o AG combinar a entrada correspondente da tabela verdade na posição j, o valor um é adicionado para $f_j(x)$, em outro caso nenhum valor é adicionado). A função $w(x)$ retorna um inteiro igual ao número de *WIRES* (fios/operação nula) presentes no circuito x, note que esse valor é adicionado à função de *fitness* apenas se $f(x)$ for factível [24].

3.1.3 Resultados

Os resultados foram comparados entre modelos de circuitos projetados por projetistas humanos, representação binária convencional (BGA) e a proposta do trabalho, chamada de NGA (mostrados nas Tabelas 3.2, 3.3 e 3.4), a qual usa uma codificação de cardinalidade n. A Tabela 3.1 apresenta o comportamento da função desse exemplo, para o qual foram usadas quatro entradas e uma saída. O tamanho da matriz é 5x5. Por fim, nota-se que a solução produzida pelo NGA é bastante atípica uma vez que utiliza uma negação no final da expressão booleana.

O resultado encontrado pelo NGA para este exemplo obteve a expressão $f = (((W \oplus Wx) \oplus ((Z + X + Y) \oplus Z)))'$, com um circuito de 8 portas, sendo uma AND, três ORs, três XORs e uma NOT, como visto na Tabela 3.2.

Para representação binária convencional (BGA) o resultado do número e tipo de portas foi idêntico ao NGA proposto, exceto a expressão descrita $F = (Z \oplus ((W \oplus Y) + XY)) \oplus$

Tabela 3.1: *Tabela verdade para o exemplo apresentado.*

W	X	Y	Z	**F**
0	0	0	0	1
0	0	0	1	1
0	0	1	0	0
0	0	1	1	1
0	1	0	0	0
0	1	0	1	0
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	0

Tabela 3.2: *Resultados para o exemplo produzidos pelo NGA.*

NGA
$F = (((W \oplus Wx) \oplus ((Z + X + Y) \oplus Z)))'$
8 portas
1 AND, 3 ORs, 3 XORs, 1 NOT

$(Z + (X + Y))'$, como visto na Tabela 3.3. Já o resultado encontrado por um humano projetista de circuitos, apresentado na Tabela 3.4, foi mais "caro". Utilizou-se de 11 portas, quatro ANDs, uma OR, duas XORs e quatro NOTs. Resultando na expressão $F = ((Z'X) \oplus (Y'W'))(Z \oplus W'))$.

3.2 Síntese de Redes Lógicas Multiníveis de Custo Mínimo Via Algoritmo Genético

Como proposta de otimização de circuitos, Barry e.t*al* [27] desenvolveram um projeto para o problema de síntese de custo mínimo utilizando também AG's. Se tratando de codificação, verificou-se que o projeto traz um método interessante e que pode ser explorado.

O artigo descreve uma abordagem ao lançar o problema de síntese de circuitos referente à lógica de custo mínimo com técnicas de AG. Descreve também um meio para acelerar a

Tabela 3.3: *Resultados para o exemplo produzidos pelo BGA.*

BGA
$F = (Z \oplus ((W \oplus Y) + XY)) \oplus (Z + (X + Y))'$
8 portas
1 AND, 3 ORs, 3 XORs, 1 NOT

Tabela 3.4: *Resultado para o exemplo produzido por um projetista humano.*

Projetista Humano
$F = ((Z'X) \oplus (Y'W'))(Z \oplus W'))$
11 portas
4 ANDs, 1 OR, 2 XORs, 4 NOTs

velocidade do AG por meio de uma função de custo implementada em hardware [27]. A tarefa de simplificação da função lógica é dividida em minimização lógica de dois níveis e minimização lógica multinível. A evolução é primeiramente impulsionada pela correção relativa do circuito, então, após os circuitos corretos terem sido gerados, a evolução é impulsionada pelo custo dos circuitos.

3.2.1 Representação do Cromossomo

De acordo com a Figura 3.3, o cromossomo representa uma matriz de conexão de portas NOR e entradas primárias para a função lógica. Um (1) no cromossomo representa uma conexão. A saída do circuito é definida como sendo a mais a direita na matriz de ligação.

Uma matriz desse formato pode descrever qualquer rede que não incorpore retorno e que contenha elementos semelhantes de circuitos (nesse caso portas NOR). As linhas da matriz de conexão representam entradas de função (a parte superior retangular da matriz) ou portas de saídas (a parte inferior triangular da matriz). As colunas representam as entradas para as portas.

Um (1) na matriz de conexão representa uma conexão do sinal representado pela linha e a porta pela coluna da matriz. A matriz é um método dinâmico, capaz de descrever circuitos de qualquer tamanho e permite que entradas possam ser ligadas a qualquer uma das portas do circuito ou vice-versa.

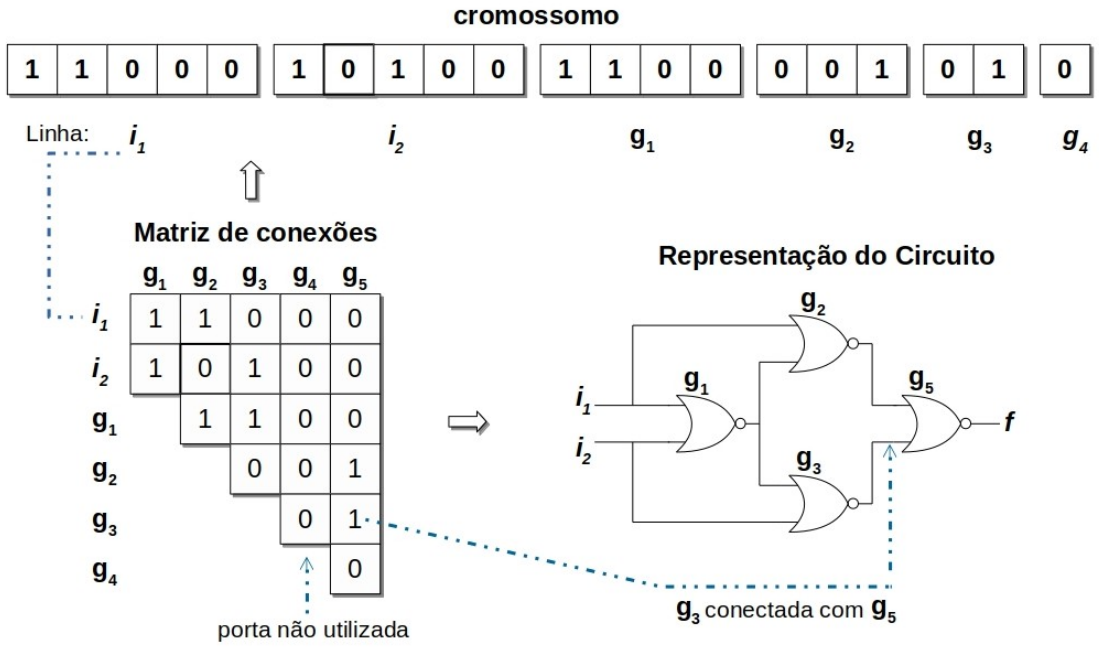


Figura 3.3: *Formato de dados do cromossomo: A matriz de conexões e representação de circuito utilizando apenas portas NOR.*

3.2.2 Função de Fitness

Para o problema de minimização lógica, uma maior aptidão está associada à um menor custo, por isso os autores apresentam uma função de custo direta. Neste trabalho, o custo da rede lógica é definido como a soma dos custos das portas individuais que compõem a rede.

Por exemplo, uma porta NOT tem um custo de duas *Basic Cells*[1] (BCs) e uma NAND ou NOR de 3 entradas tem um custo de 4 BCs. Se o *fan-in*[2] de uma porta é zero, então seu custo também é zero. No entanto, ainda é necessário considerar o custo de redes lógicas produzidas pelo AG em uma determinada iteração, que não implementam corretamente a função lógica fornecida. Para essas redes, é adicionado um incremento de custo como penalidade para cada instância em que a rede lógica F não fornece a saída especificada pela função de origem T quanto testada contra todas as possíveis combinações de entrada.

Ao escolher o incremento de penalidade como sendo o custo máximo possível (isto é, $n_c + n_g$) para uma matriz de conexão, podemos ter certeza de que uma rede com n_e erros,

[1]Definição dada por Barry [27] para a quantidade de entradas (*fan-in*) de uma porta lógica somada com um (*fan-in + 1*).

[2]Termo que define o número máximo de entradas digitais que uma porta lógica pode aceitar [28].

terá um custo menor do que uma rede com $n_e + 1$ erros [27]. Portanto, o custo C para um determinado cromossomo é composto por um custo de penalização acrescentado ao custo intrínseco da rede.

3.2.3 Resultados

Os resultados para este trabalho se dá através das experiências de síntese realizadas em duas funções lógicas. A primeira, uma função de paridade ímpar de três bits e a segunda, um comparador de magnitude de dois bits. A técnica de implementação em todos os casos foi uma biblioteca composta apenas de portas NOR com o custo de porta em células básicas (BCs) sendo calculado como o *fanin* da porta mais 1.

Foi utilizado uma ferramenta de lógica de síntese da companhia Synopsys chamada de *Synopsys Design Compiler* [29] para comparação dos resultados com o AG como demostrados nas Figuras 3.4 e 3.5.

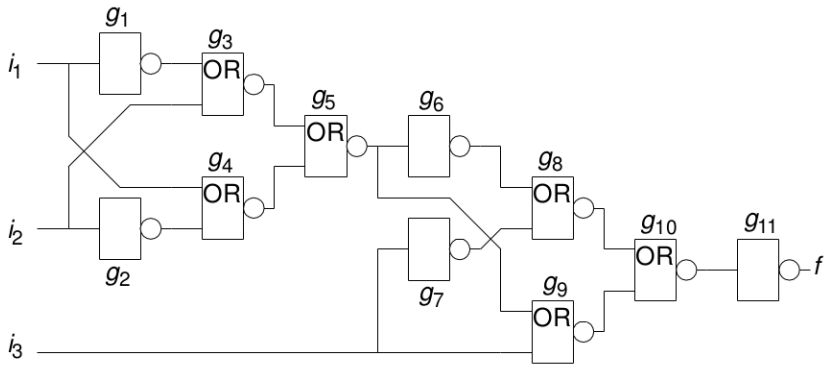


Figura 3.4: *Circuito gerado pelo Synopsys Design Compiler com custo de 28 BCs.*

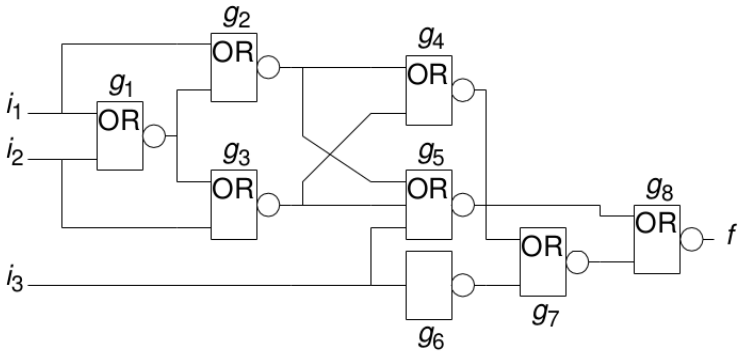


Figura 3.5: *Circuito gerado pelo AG com custo de 24 BCs.*

3.3 EHW - Aplicado à Síntese de Circuitos Digitais Usando Representação por Portas Lógicas

A proposta de Sobrinho *et.al* [30] está baseada no conceito de circuitos evolutivos, onde é proposto uma metodologia para síntese e otimização de circuitos digitais combinacionais de baixa e média complexidade, utilizando o AG.

3.3.1 Representação do Cromossomo

A opção adotada para o cromossomo nesse caso, foi por uma codificação de números inteiros e em forma matricial linear numérica, que permite também a variação do comprimento do cromossomo de modo a se obter um circuito correto e minimizado.

A codificação da Tabela 3.5 indica o significado para cada gene. Na coluna 1, os três primeiros bits (da esquerda para a direita) definem a porta a ser utilizada, e o quarto bit determina como serão realizas as conexões da porta (assim como na abordagem de [24], restringe-se o modelo à portas de duas entradas). O número decimal associado é mostrado na coluna 2, e na coluna 3 os tipos de portas que estão sendo utilizadas [30].

	Níveis do circuito				
	1	**2**	**3**	**4**	
Entradas		$S_{i,j-1}$	$S_{i,j}$		**Saídas**
		$S_{i+1,j-1}$			

Tabela 3.5: *Representação Matricial do Circuito.*

Ainda na Tabela 3.5 observa-se que cada elemento da matriz bidimensional (utiliza a mesma proposta de [24]) é uma porta lógica limitada a no máximo duas entradas ou fio de ligação. De uma maneira mais formal, uma porta S na sua posição S_i, onde j indica o nível da porta (coluna), uma entrada é proveniente da posição $S_{i,j-1}$ (coluna anterior e uma linha acima). Com conexão para os extremos, ou seja, para elementos da primeira $S_{1,j}$ e última linha $S_{n,j}$. Nestas condições, para os elementos da primeira linha, a primeira entrada vem da posição $S_{1,j-1}$ e a segunda da posição $S_{n,j-1}$. Para elementos da última linha $S_{n,j}$, a primeira entrada vem da posição $S_{n,j-1}$, e a segunda é obtida de qualquer posição (uma representação pode ser vista na Figura 3.1) $S_{1,j-1}$ para $i = 1, ..., n$ e $j = 1, ..., k$, onde n e k são números inteiros e não necessariamente diferentes um do

outro [30].

3.3.2 Função de Fitness

A função de avaliação é construída em duas etapas. A primeira perfaz a soma de acertos da sequência de saída de uma tabela verdade ou expressão booleana, de acordo com:

$$f(x) = \sum_{i}^{l} x_i \tag{3.2}$$

onde, a função $f(x)$ indica o número de acertos na tabela verdade do caso sobre estudo; $x_i = 0$, falso; $x_i = 1$, verdadeiro e l é o número de linhas da tabela verdade $i = 0, 1...l..$ Em sequência, na segunda etapa, tendo o circuito que execute a tabela verdade (factível), é iniciado o processo de minimização em termos de entradas das portas e o número de portas lógicas no circuito. Para isso trocam-se as portas lógicas por fios, até que o desempenho especificado pela tabela verdade do circuito seja mantido. Desta forma, com o aumento do número de fios, obtém-se uma função $w(x)$, a qual é um valor inteiro igual à quantidade de fios adicionados ao circuito. Este procedimento de adicionar fios no lugar de componentes equivale a minimizar o circuito em termos de portas lógicas [24]. A função de adaptação, f_a para este caso é escrita como:

$$fa(x) = f(x) + w(x) \tag{3.3}$$

3.3.3 Resultados

Um dos exemplos dados pelos autores mostrado a seguir, fazem comparações com o método de minimização de mapa de Karnaugh.

Como mostrado na Tabela 3.6 o resultado é proveniente de um circuito combinacional de 3 entradas e uma saída, onde a população inicial é gerada de forma aleatória e uma matriz 5x5, ou seja, o tamanho do cromossomo (sequência numérica) de 25 genes. Os parâmetros de controle utilizados são: uma população de tamanho 500, número máximo de iterações de 500, taxa de recombinação de 50%, taxa de mutação de 5%.

O resultado da proposta é chamado de EHW na Tabela 3.6 e produziu a expressão

EHW
$f_1 = AB \oplus [(A + B)C]$
4 portas
2 ANDs, 1 OR, 2 XORs

Mapa de Karnaugh
$f_2 = C(A \oplus B) + B(A \oplus C)$
5 portas
2 ANDs, 1 OR, 2 XORs

Tabela 3.6: *Resultados obtidos pelo método proposto e método de Karnaugh.*

$f_1 = AB \oplus [(A + B)C]$, com 4 portas, duas ANDs, uma porta OR e duas XORs. O Mapa de Karnaugh por sua vez, obteve a expressão $f_2 = C(A \oplus B) + B(A \oplus C)$, com 5 portas, sendo duas ANDs, uma porta OR e duas portas XOR.

3.4 Síntese de Circuitos Digitais Utilizando Computação Evolutiva

Lacerda e.t*al* [31] propõe um método de síntese automática de circuitos eletrônicos digitais utilizando a técnica de computação evolutiva. Partindo de uma tabela verdade de dados binários com valores de entrada e saída digitais, o método visa encontrar a função lógica minimizada do circuito digital utilizando operadores genéticos aplicados a equações booleanas na forma de soma de produtos. A expressão booleana encontrada pode ser implementada em um dispositivo programável para posterior utilização como, por exemplo, PAL[3] (*Programmable Array Logic*) e FPGA[4] (*Field Logic Programmable Gate Array*).

A entrada do sistema é um arquivo com uma tabela verdade que descreve as entradas e saídas binárias de um circuito digital combinacional. O módulo otimizador de circuitos recebe a tabela verdade e evolui através de um algoritmo evolutivo (no caso, um AG) para o melhor circuito possível. Após essa evolução, o módulo envia somente a melhor solução do circuito encontrado para ser configurada no dispositivo de *hardware* [31]. O ciclo de funcionamento do sistema é apresentado na Figura 3.6.

[3]Dispositivo de *Hardware* de lógica programável capaz de armazenar qualquer expressão booleana na forma de soma de produtos dentro dos seus limites de tamanho [31].

[4]Dispositivo de lógica programável que implementa lógica multinível. Define-se como uma matriz de células lógicas colocadas em uma infra-estrutura de interconexões que podem ser programadas em três níveis: função das células lógicas, interconexão entre as células e entradas e saídas. FPGAs são dispositivos altamente versáteis que oferecem ao usuário uma ampla gama de opções de design [32] [33].

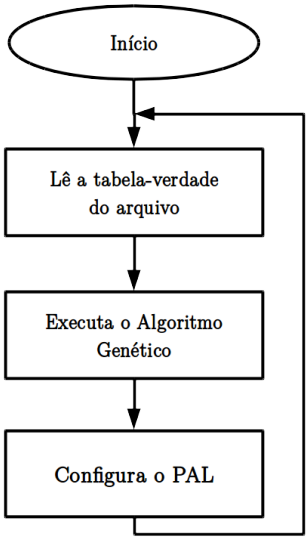


Figura 3.6: *Fluxograma do sistema.*

A seguir na Figura 3.7 é apresentado um exemplo da função lógica XOR de 2 entradas no formato de um arquivo de entrada (matriz) com os valores de entrada (coluna 0) e saída (coluna 1). A seção seguinte explica com mais detalhes a estrutura de um arquivo de entrada.

```
.i   2
.o   1
.p   4
00   0
01   1
10   1
11   1
.e
```

Figura 3.7: *Exemplo 1: Estrutura do arquivo de entrada (matriz).*

3.4.1 Representação do Cromossomo

A codificação de cada indivíduo foi feita utilizando uma matriz, onde cada linha representa um termo de produto e cada posição dessa linha representa uma entrada da tabela verdade.

Por exemplo, o valor de entrada negada é representado por 0 e, para entradas não negadas, é representado pelo valor 1. A ausência de variável *don't care* é representada por "-".

No exemplo da função XOR na seção anterior, cada linha da matriz possuirá duas posições uma vez que a tabela verdade para o exemplo possui duas entradas. O número máximo de linhas para qualquer indivíduo é o número de linhas que a tabela possui cuja saída é 1. Desta forma, um exemplo para a função booleana $(\overline{A}B) + \overline{B}$ é apresentado na Figura 3.8.

```
01
-0
```

Figura 3.8: *Exemplo 2: Estrutura do arquivo de entrada (matriz).*

3.4.2 Função do Fitness

A função do *fitness* apresentada possui dois objetivos:

- valorizar os indivíduos que acertam mais linhas da tabela verdade;
- valorizar os indivíduos menores (menos termos de produto).

Segundo os autores, esses dois objetivos apresentados são muitas vezes conflitantes, por isso uma boa função de avaliação deve ser capaz de pondera-los. Para cada objetivo, foi apresentado uma função. São elas:

$$f_1 = 100 \div 2^i \times \sum_{j=0}^{2^i-1} (1 - (x_j - d_j)) \tag{3.4}$$

onde i é o número de entradas do sistema, e cada j representa uma combinação das entradas, x é a saída obtida pelo indivíduo para a combinação j das entradas. Já para o segundo objetivo foi utilizada a seguinte expressão:

$$f_2 = 1 \div (100 + tam) \tag{3.5}$$

onde *tam* é o número de termos de produto do indivíduo. A constante 100 somada no denominador foi obtida através de testes e seu valor foi o que proporcionou melhores resultados [31]. Por fim, o *fitness* é a diferença da soma das duas funções:

$$fitness = -(f_1 + f_2) \tag{3.6}$$

Onde o objetivo é encontrar o indivíduo com o menor valor de *fitness*.

3.4.3 Resultados

O problema escolhido, função lógica XOR, tem como objetivo encontrar a melhor função booleana que representa a função lógica XOR para uma quantidade arbitrária de entradas binárias e uma saída digital [31]. De acordo com os autores, o AG desenvolvido foi capaz de encontrar o circuito otimizado referente a porta XOR através de soma de produtos.

Apesar de citar outros testes de sucesso realizados para a função lógica OR, AND, NAND, somador binário completo e tabelas arbitrárias com até quatro entradas, os autores não apresentam nenhum gráfico comparativo ou tabela com detalhes dos resultados.

4 Metodologia

Neste capítulo é descrita as técnicas aplicadas na construção do Algoritmo Genético (GA) para otimização de circuitos combinacionais. O objetivo desse capítulo é apresentar desde o modelo de código intermediário e o processo de leitura, até os operadores genéticos e os métodos utilizados para o desenvolvimento.

O capítulo foi divido em seis seções principais, que descrevem sobre as etapas do processo de otimização, apresentação de um modelo do código intermediário estruturado em JSON, as etapas de codificação do Algoritmo Genético, o método de avaliação utilizado e por fim os diagramas UML.

4.1 Métodos utilizados

Após estudos sobre o tema, alguns trabalhos relacionados foram estudados para aprofundamento do conteúdo com o intuito de encontrar um bom roteiro para a codificação, estruturação e adequação do Algoritmo Genético para o tipo de problemática que este trabalho apresenta. Como resultado desse processo, nos tópicos seguintes é feita uma breve descrição de alguns dos métodos adotados para o desenvolvimento do trabalho.

- **Código intermediário JSON**: Com o objetivo de flexibilizar a utilização dos modelos de circuitos processados pelo algoritmo em qualquer outra ferramenta ou em qualquer outro tipo uso, o código intermediário JSON neste trabalho, representa a entrada e a saída após o processamento da aplicação, como mostra a Figura 4.1, sendo ambos os arquivos escritos no mesmo padrão do exemplo da Figura 4.2, o que muda no caso, é a expressão booleana em função da operação NOR do arquivo de saída.

- **Uso de um único tipo de porta lógica**: Um circuito combinacional é comumente projetado utilizando vários tipos de portas lógicas, porém algumas delas podem ter um custo maior que outras, caso estivermos nos referindo a fabricação do circuito. Graças a universalidade das portas NAND e NOR é possível construir um circuito combinacional equivalente utilizado no arranjo do circuito apenas uma das duas

portas, garantindo assim um modelo de menor custo que implementa a mesma função de um outro que utiliza diferentes tipos de portas. Neste trabalho a expressão booleana do arquivo de saída vem escrita em função de portas do tipo NOR.

- **Utilização de matriz**: A estrutura de uma matriz de valores binários pode arranjar qualquer circuito combinacional onde uma entrada (linha) pode ser ligada a qualquer porta (coluna) e uma porta (linha) a qualquer outra (coluna) dentro de um circuito. Nesse caso uma ligação é definida pelo bit '1'. O vetor cromossomo é construído através dessa matriz de conexões e por consequência as duas estruturas possuem o mesmo tamanho. Por essa razão o tamanho do cromossomo varia de acordo com o número de variáveis que a função possui.

- **Tabelas verdade**: Uma vez lido o JSON, a tabela verdade para o circuito de entrada do GA é criada, da mesma maneira, cada indivíduo da população possui a sua. As tabelas servem para comparar a saída da função de cada indivíduo com a função do circuito de origem. Através dessa comparação é possível saber se o indivíduo em questão apresenta solução factível ou não.

- **População aleatória**: Indivíduos da primeira geração são definidos randomicamente. A quantidade é definida de acordo com o número de entradas e é fixa, ou seja, não sofre alteração de tamanho ao longo da execução.

- **Processo de evolução**: O GA evolui em duas etapas, a primeira com foco no resultado da função de cada indivíduo, e a segunda com foco nos custos de fitness.

- **Avaliação do circuito**: Este trabalho utiliza o cálculo de células básicas da tecnologia CMOS e compara o valor do resultado do GA com o valor dos resultados obtidos pelo método de Karnaugh e Quine-McCluskey.

4.2 Código intermediário estruturado em JSON

JSON é um formato de texto que facilita a troca de dados entre todas as linguagens de programação. Sua sintaxe utiliza apenas alguns poucos símbolos que são: chaves, colchetes, dois pontos e vírgula. Essa estrutura simples é um dos aspectos que torna o JSON uma linguagem atrativa em várias aplicações.

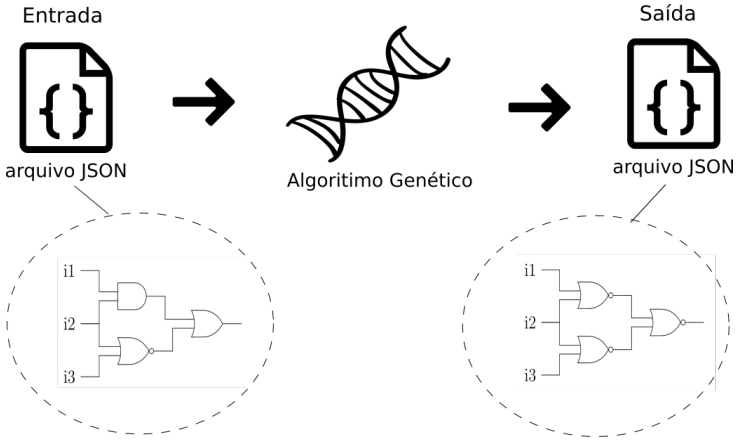


Figura 4.1: *Processo do projeto de otimização*

Muitas linguagens utilizam a notação de objetos, contudo isto não é um padrão absoluto. Modelos podem divergir drasticamente, dependendo da linguagem. O JSON não trabalha com objetos de fato, ao invés disto pares de informações, rótulo do objeto e valor, são armazenados. Ambos são cadeias de caracteres ou *Strings* [34].

O JSON também dá suporte a listas de valores, que são criadas através do uso de colchetes. Todo elemento incluso no intervalo dos colchetes é um elemento da lista. O aninhamento de vetores e objetos permite a representação de estruturas de dados mais complexas como árvores [34].

Neste trabalho a notação JSON é utilizada para a modelagem de um código intermediário para uma etapa de otimização de circuitos combinacionais. Na Figura 4.2 é listado um exemplo do padrão de código em JSON que será utilizado no trabalho.

No exemplo ilustrado na Figura 4.2 são descritos dois circuitos distintos, *circuito_1* e *circuito_2*. Dentro do escopo de cada circuito são descritas as saídas e entradas relacionadas em *outputs* e *inputs*, respectivamente. Adicionalmente, existem os objetos *expression* e *minimized_expression* representando a expressão booleana original e após o processo de minimização. O comportamento do circuito é determinado pela função booleana de cada saída existente para um mesmo circuito.

4.3 O Algoritmo Genético

Nesta seção é descrita as técnicas e configurações utilizadas para o desenvolvimento do Algoritmo Genético (GA) de otimização de circuitos digitais. Desenvolvido na linguagem de programação Java.

```
{
  "circuitName": "maioria",
  "inputs": [
    {
      "name": "a"
    },
    {
      "name": "b"
    },
    {
      "name": "c"
    }
  ],
  "outputs": [
    {
      "expression": "b c + a c + a b",
      "minimizedExpression": "(a + b) (a + c) (b + c)",
      "name": "result"
    }
  ]
}
```

Figura 4.2: *Exemplo de código intermediário JSON (oriundo da etapa de compilação) de entrada da etapa de otimização, a qual este trabalho implementa*

Entre os trabalhos da seção "Trabalhos Relacionados" destaca-se o de Barry [27] por apresentar um objetivo muito semelhante ao que este trabalho propõe, em consequência o modelo do algoritmo é replicado em grande parte.

Uma vez lida as variáveis de entrada do arquivo JSON do circuito que se deseja otimizar, chamado de C_{or} (refere-se ao circuito de entrada, origem) e sua expressão booleana (ver exemplo na Figura 4.2), a tabela verdade para o modelo é criada para comparação com as tabelas verdade de cada indivíduo da população tornando possível a classificação de factíveis e não factíveis. O número de indivíduos da população n_p (ver Equação 4.2), a taxa de mutação m_p, a taxa de *crossover* c_p, são os três parâmetros definidos para o GA como podemos observar no Algoritmo 1.

Após gerar a população inicial o GA implementa uma maneira de proteger sempre o melhor indivíduo presente em cada geração. Inicialmente guarda na variável *melhor* o cromossomo do índice zero da população (em ordem, os melhores fitness ocupam as primeiras posições) e a cada iteração, uma comparação do fitness do indivíduo dessa posição é realizada com o fitness do *melhor*. Caso um melhor custo seja encontrado, a variável é atualizada.

O processo de evolução é realizado em duas etapas. A primeira é responsável por deixar todos os indivíduos da população factíveis para posteriormente na segunda, evo-

Algorithm 1 Algoritmo Genético

```
1: function GA(n_p, m_p, c_p)
2:     Generate initial population
3:     for i ← 1 to n_p do
4:         cromossom[n_c] = Random()
5:         population[i] = cromossom
6:     end for
7:     best = population[0]
8:
9:     First evolution
10:    while population[n_p − 1] notfactivel do
11:        crossover(c_p)
12:        orderPopulation()
13:    end while
14:    bestFitness = population[0].fitness
15:
16:    Second evolution
17:    while bestFitness < population[n_p/2].fitness do
18:        crossover(c_p)
19:        orderPopulation()
20:        if population[0].fitness < best.fitness then
21:            best = population[0]
22:        end if
23:    end while
24:    return(best)
25: end function
```

luir seus custos, ou seja, a evolução é conduzida pelo custo do circuito. O paradigma de sobrevivência dos mais aptos (baixo custo) em que os filhos mais aptos substituem aleatoriamente os indivíduos menos aptos, assegura o avanço evolutivo para uma solução ótima [27].

4.3.1 População Inicial

Uma população de n_p cromossomos é gerada aleatoriamente, no entanto é importante definir uma quantidade ideal de indivíduos de acordo com o tamanho do circuito de entrada, isto é, o espaço de soluções que o algoritmo pode trabalhar. Segundo Barry [27], o número de diferentes funções lógicas (tabelas verdade) aumenta exponencialmente de acordo com o número de variáveis de entrada *(inputs)*, e o número de funções lógicas possíveis n_f para uma função binária de n_i entradas é dado por:

$$n_f = 2^{2^{n_i}} \tag{4.1}$$

Por exemplo, para um circuito de três entradas temos apenas duzentos e cinquenta e seis funções possíveis e para um circuito de cinco entradas esse número salta para mais de quatro bilhões de funções possíveis, ou seja, cresce de forma muito rápida. Usar uma população desse tamanho é inviável, o GA se tornaria lento demais para a execução, principalmente em máquinas convencionais. Devido a esse problema, o tamanho ideal para a população n_p, é dado por:

$$n_p = \left(inputs^2 + 1\right)^2 \tag{4.2}$$

O número de entradas *inputs* ao quadrado mais um, é o mesmo que o número de portas lógicas do circuito elevado ao quadrado. Por exemplo, a quantidade de indivíduos da população para um circuito de três entradas, é cem.

A equação 4.2 não é a única maneira para se definir o tamanho da população no GA. O parâmetro n_p (número da população) visto no Algoritmo 1, permite que a população obtenha qualquer tamanho.

4.3.2 Matriz de conexões de portas NOR

A utilização da matriz de conexões é crucial para auxiliar no processo de definição do cromossomo e cálculo do fitness. A matriz da Figura 4.3 é definida de modo que qualquer entrada da função pode ser ligada a qualquer porta e qualquer porta lógica g_n pode ser conectada a qualquer outra entrada g_k em que $k > j$. Uma vez completa, a matriz de conexão é capaz de descrever todos os circuitos de até n_g portas e n_i entradas da função. O tamanho da parte superior (retangular) da matriz é dado por n_i . n_g. Já o tamanho da porção inferior (triangular) é formado por $n_g(n_g - 1)/2$ células [27].

As linhas i_n representam entradas de função (parte superior retangular da matriz) ou portas de saídas (parte inferior triangular da matriz). As colunas são representadas pelas portas. As entradas estão representadas por i_n e saídas por f. Não há restrições quanto ao número de entradas e saídas. O modelo suporta n entradas sendo $n \geq 2$ e n saídas onde $n \geq 1$.

4.3.3 Cromossomo

Neste trabalho um cromossomo representa um circuito digital. Dessa maneira, ao longo do texto os termos, indivíduo, cromossomo ou rede lógica, são análogos à circuitos digitais.

Os genes são representados tradicionalmente por zeros e uns. O tamanho de um cromossomo n_c é definido pela Fórmula 4.3, onde o número de entradas é representado por n_i e o número de portas por n_g. Logo, o tamanho do cromossomo depende do número de entradas e portas.

$$n_c = n_i n_g + \frac{n_g(n_g - 1)}{2} \tag{4.3}$$

O formato do vetor binário *cromossomo* pode ser visto da Figura 4.3. Os genes são distribuídos no vetor de acordo com o percurso em ordem da matriz de conexões, onde as linhas referentes à entradas i_1, i_2,..., i_n, são arranjadas nas primeiras posições do vetor cromossomo precedidas pelas linhas da matriz referentes às portas lógicas. Note que a porta de saída (coluna à direita da matriz) não aparece no cromossomo porque é a porta de saída do circuito e não pode ser ligada a nenhuma outra porta de acordo com a configuração da matriz de conexões.

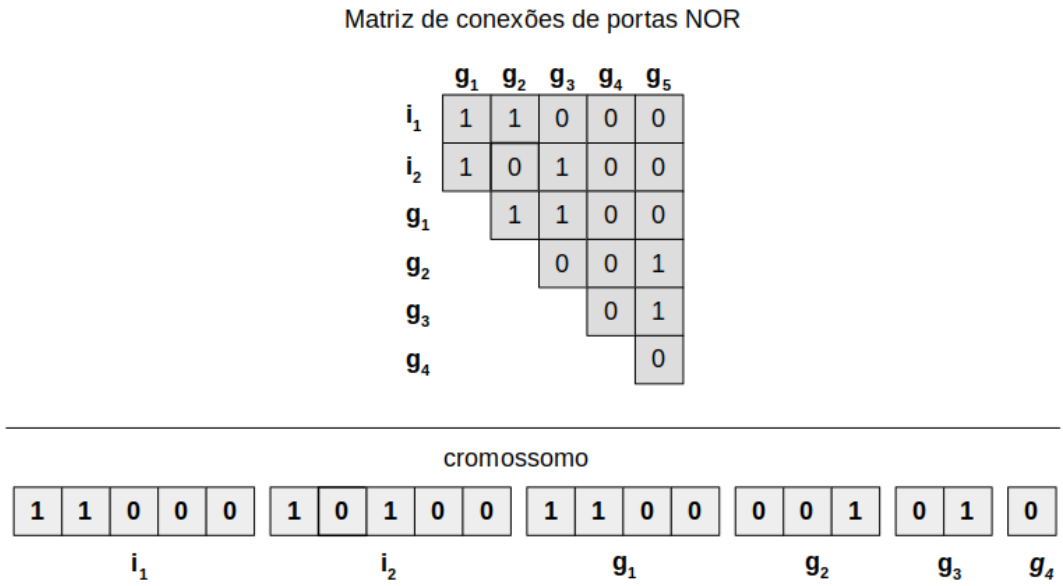


Figura 4.3: *Cromossomo formado a partir da matriz de conexões.*

Portas lógicas que possuem uma ligação (entrada), são representadas pelo bit '1'. Quando não existe uma ligação, então o bit que representa esse caso é '0'. Na matriz da Figura 4.3 por exemplo, a entrada i_1 está ligada com a porta g_2 assim como a porta g_1 está ligada com a porta g_3.

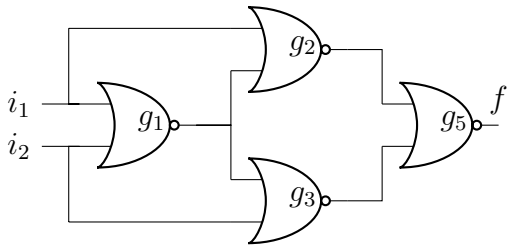


Figura 4.4: *Circuito da matriz de conexões da Figura 4.1 representado com portas NOR.*

A Figura 4.4 implementa o circuito descrito na matriz da Figura 4.3 representado por portas lógicas NOR. A expressão booleana é descrita como: $\overline{\overline{A + \overline{A + B}} + \overline{B + \overline{(A + B)}}}$.

4.3.4 Função do Fitness

Quando se trata do problema de minimização de circuitos digitais, um indivíduo factível que detém um fitness alto, é aquele que possui um custo baixo. Quanto menor o número de portas do circuito, menor é o custo dele. Para ser factível o indivíduo precisa implementar a função do circuito de entrada. Em outras palavras, as saídas da função do circuito em questão são iguais quando comparadas às saídas da função do circuito C_{or}.

O GA mede o custo de um indivíduo em *basic cells* (BCs), através da soma dos custos individuais de cada porta lógica do circuito. Uma porta NOT (possui apenas uma entrada) tem um custo de duas BCs e uma NAND ou NOR de três entradas tem um custo de quatro BCs. Se o *fan-in* de uma porta for zero, então seu custo também é zero. Para portas NAND e NOR, o custo C_B de uma porta g em BCs é dado por:

$$C_B(g(fan-in)) \begin{cases} fan-in+1 & se \quad fan-in \geq 1 \\ 0 & caso \quad contrário \end{cases} \tag{4.4}$$

Por exemplo, se um circuito factível possui quatro portas NOR com duas entradas cada uma, seu custo então é doze. No entanto, o GA produzirá indivíduos não factíveis, ou seja, que não implementam corretamente a função lógica de origem C_{or} e por isso é adicionado um incremento de custo como penalidade para cada instância em que a rede lógica F não fornece a saída especificada pela função de origem T quando testada contra todas as possíveis combinações de entrada [27]. Essas comparações são feitas utilizando as tabelas-verdade de T e F.

Ao escolher o incremento de penalidade como sendo o custo máximo possível (isto é, $n_c + n_g$) para uma matriz de conexão, podemos ter certeza de que uma rede com n_e erros, terá um custo menor do que uma rede com $n_e + 1$ erros [27]. O custo da penalidade é dado por:

$$C_p = (n_c + n_g) \sum_{i=0}^{2^{n_i}-1} \begin{cases} 1 & se \quad T(i) \neq F(i) \\ 0 & caso \quad contrário \end{cases} \tag{4.5}$$

Portanto, o custo C para um determinado cromossomo é composto por um custo de penalização acrescentado ao custo intrínseco da rede:

$$C = C_p + \sum_{i=1}^{n_g} C_B(g_i) \tag{4.6}$$

4.3.5 Crossover e Mutação

A taxa de cruzamento, também chamada de percentual de crossover c_p define a quantidade de indivíduos da população que doa seu material genético para a geração de filhos. De acordo com o Algoritmo 2, a seleção dos pais é feita de forma aleatória, assim o GA sorteia um par de cromossomos (pais) distintos para geração de um único filho.

Algorithm 2 Algoritmo Genético

```
1: function CROSSOVER(c_p)
2:     for i ← 1 to c_p do
3:         parent1 = population.Random()
4:         parent2 = population.Random()
5:         if parent1.fitness > parent2.fitness then
6:             positionWorst = population[parent1.index]
7:             fitnessWorst = population[parent1.fitness]
8:         else
9:             positionWorst = population[parent2.index]
10:            fitnessWorst = population[parent2.fitness]
11:        end if
12:    end for
13:    cutPosition = Random()
14:    for i ← 0 to cutPosition do
15:        offspring = parent1.genes
16:    end for
17:    for i ← cutPosition to chromossom.size do
18:        offspring = parent2.genes
19:    end for
20:    mutacao(offspring, m_p)
21:    if offspring.fitness > fitnessWorst then
22:        population[positionWorst] = offspring
23:    end if
24: end function
```

Posteriormente o Algoritmo escolhe o pai que possui o maior custo do fitness entre os dois e guarda na variável chamada de *posicaoPior* sua posição (index) do vetor da população. Já o custo de fitness desse pior indivíduo é armazenado na variável chamada de *fitnessPior*.

Utiliza-se apenas um ponto de corte. Esse ponto único é definido também de forma aleatória garantindo que não seja escolhido a primeira posição do cromossomo e nem a última. De acordo com a Figura 4.5, a primeira metade do pai 1 compõe a primeira metade do filho gerado enquanto a segunda metade do pai 2 compõe a segunda metade do filho.

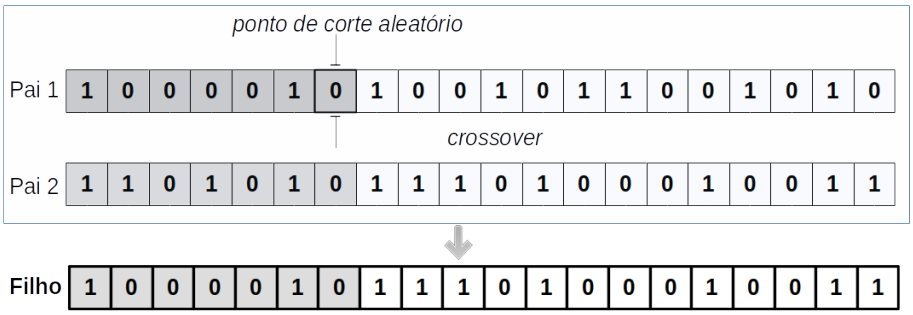


Figura 4.5: *Função de cruzamento*

Após o cálculo do fitness do filho ser realizado, o valor é comparado com o valor que está armazenado na variável $fitnessPior$. Caso o fitness do filho seja pior que o pior fitness entre os dois pais, a inserção do filho na população no local armazenado em *posicaoPior* não é realizada, garantindo que a população possa receber sempre melhores indivíduos mantendo assim a evolução em cada geração.

No caso da mutação, ela acontece apenas no filho (como podemos ver na Figura 4.6) todas as vezes que eles são gerados. Importante ressaltar que a mutação acontece antes do fitness do filho ser comparado com o fitness do pior pai. Uma vez definido o percentual de mutação m_p, a quantidade de bits que será alterada no vetor cromossomo é a mesma para todos os indivíduos até o final da execução do GA.

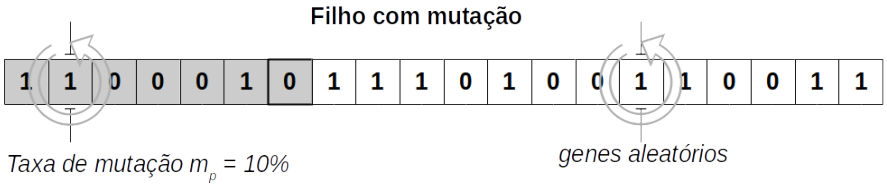


Figura 4.6: *Função de mutação*

4.3.6 Critério de Parada

Observa-se no Algoritmo 1 que o critério de parada é tratado durante a segunda evolução do GA. Quando a primeira evolução é finalizada, ou seja, quando todos os indivíduos forem factíveis, a variável *melhorFitness* recebe o valor do melhor fitness gerado até então. Sendo assim, a condição de parada do *while* (linha 17) é satisfeita quando o

valor fitness do indivíduo que ocupa a posição na metade da população $n_p/2$ for maior que o valor de *melhorFitness*. Esta é uma abordagem boa e garante que um indivíduo com ótimo valor de aptidão seja encontrado porém, pode levar muito mais gerações para finalizar do que se não implementasse um critério de parada.

4.4 Método de avaliação

Com a realização dos testes, procurou-se uma maneira de comparar os modelos dos circuitos minimizados pelo GA com os resultados obtidos pelo trabalho de Barry, pelo mapa de Karnaugh e Quine McCluskey. As células básicas (BCs) apresentada por Barry [27] é utilizada neste trabalho para avaliar e comparar o custo dos resultados obtidos pelo GA dos outros métodos apresentados. A avaliação dos circuitos testados em (BCs) da tecnologia *CMOS*[1] é relevante, pois essa é a tecnologia mais utilizada na fabricação de circuitos integrados por oferecer um baixíssimo consumo de energia.

Como descrito anteriormente na Equação 4.4, o custo de uma porta lógica em BCs é dado pela soma da quantidade de *fan-in* (entradas) que a porta possui somado mais um. Por exemplo, uma porta NOT tem um custo de duas BCs e uma NAND ou NOR de três entradas tem um custo de 4 BCs [27].

4.5 Diagramas de Classe

Nesta seção é apresentado os diagramas de classe UML modelado para a ferramenta de otimização de circuitos. Gerados separadamente, cada pacote possui seu diagrama de classe. O pacote *Entities* mostrado na Figura 4.7 é composto de quatro classes: *Chromosome*, *Gene*, *Gate* e *Element*. Já o pacote *Controller* possui apenas a classe *Genetic*. Os pacotes *Circuit JSON* e *Circuit Entities* são composto de duas e três classes respectivamente.

4.5.1 Pacote *Entities*

A classe *Chromosome* da Figura 4.7 estrutura o indivíduo (cromossomo) com todos os atributos necessários para descrever as características de um indivíduo modelado de

[1] *Complementary Metal Oxide Semiconductor* (Semicondutor de Óxido-Metal Complementar), tem como características principais o reduzido consumo de corrente (baixa potência), alta imunidade a ruídos e uma faixa de alimentação que se estende de 3v a 15v ou 18v dependendo do modelo [35].

acordo com a problemática. O atributo *fitness* armazena o resultado do cálculo do fitness para cada instância da classe, assim como *TruthTableChrom* guarda a tabela verdade utilizada para definir a função de saída do indivíduo, realizar a comparação com a tabela verdade do circuito de origem C_{or} e definir o atributo booleano *feasible* como factível ou não. A estrutura *List<Input>* por sua vez, guarda as variáveis de entrada do circuito para geração da expressão booleana final de um indivíduo em particular.

Figura 4.7: *Diagrama de classe do pacote Entities*

A relação que a classe *Chromosome* possui com as classes *Gene* e *Element* é de *0..**

(zero ou muitos). A classe *Element* é utilizada para simular o comportamento da função de uma instância do cromossomo e definir sua tabela verdade. Da mesma maneira a classe *Gate* utiliza a herança e computa as entradas de cada porta NOR individualmente. Por fim, a classe Gene possui apenas o atributo privado *connection* o qual representa o gene '1' ou '0' do cromossomo.

4.5.2 Pacote *Controller*

A classe *Genetic* é a única do pacote *Controller*. Apresentada na Figura 4.8, é a classe que implementa o construtor para o GA *Genetic()*. A população do GA é armazenada em *List<Chromosome>population* e seu tamanho (número de indivíduos) é definido pela variável do tipo *int numPopulation*.

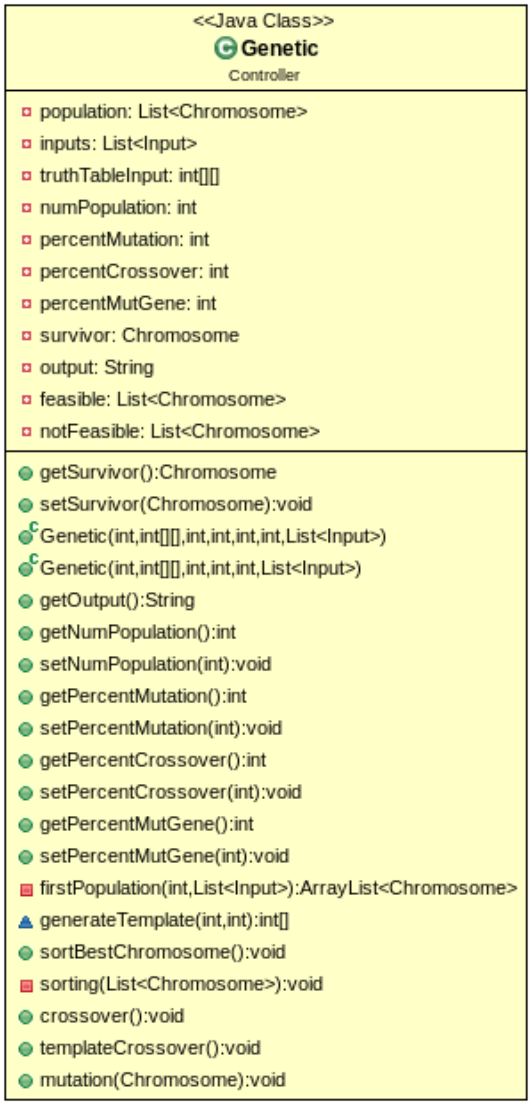


Figura 4.8: *Diagrama de classe para pacote Controller*

A taxa de mutação, taxa de crossover, a variável identificada como (*melhor*) que guarda o indivíduo de melhor fitness de uma determinada iteração durante a execução do Algoritmo, além dos métodos de crossover e mutação, são definidos nessa classe.

4.5.3 Pacote *Circuit JSON*

A leitura e escrita do arquivo intermediário JSON que representa o circuito de origem C_{or}, é implementada na classe chamada *MiddleCodeReader* da Figura 4.9. Foi utilizada a biblioteca Java Gson, para converter o arquivo JSON e um objeto Java e vice-versa.

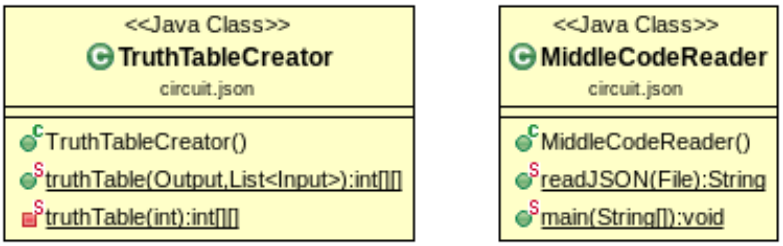


Figura 4.9: *Diagrama de classe do pacote Circuit JSON*

A classe *TruthTableCreator* implementa a criação da tabela verdade para o circuito a partir da expressão booleana descrita no arquivo JSON. Importante ressaltar que essa classe não possui relacionamento com a classe *MiddleCodeReader* por ser estática.

4.5.4 Pacote *Circuit Entities*

O diagrama de classe do pacote *Circuit Entities* mostrado na Figura 4.10 é estruturado para receber as informações do arquivo JSON de acordo com o padrão definido no início desse Capítulo apresentado na Figura 4.2.

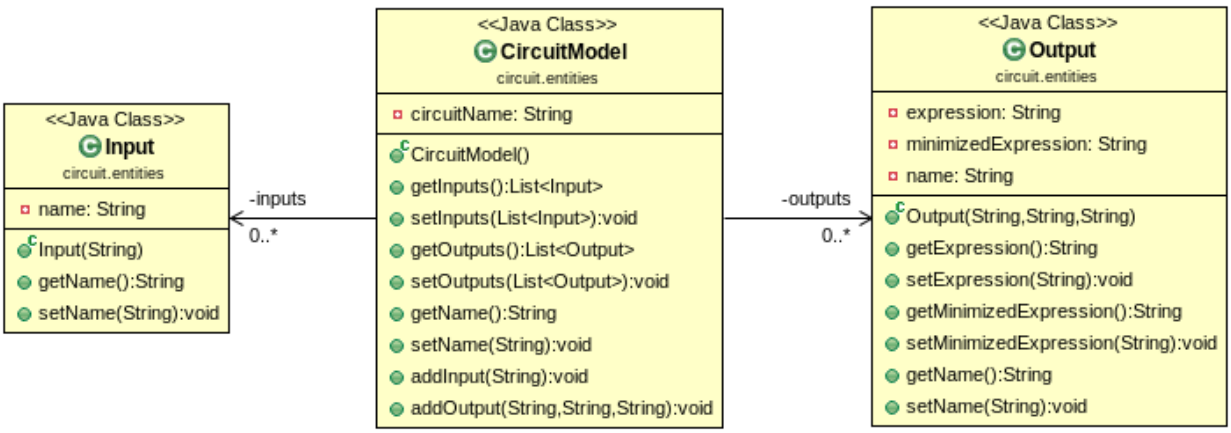


Figura 4.10: *Diagrama de classe para pacote Circuit Entities*

As classes *Input*, *CircuitModel*, *Output* é a representação do modelo do circuito em seu estado objeto. Desta forma evita que seja feita uma leitura do JSON toda vez que houver necessidade de informações do arquivo.

5 Resultados

Este Capítulo apresenta os detalhes dos três casos de testes escolhidos e seus resultados. Para cada modelo de circuito apresentado nas seções seguintes, os testes foram realizados utilizando o método de mapa de Karnaugh, Quine-McCluskey e o Algoritmo Genético (GA). Como mencionado anteriormente, a comparação entre os resultados é feita em células básicas (BCs).

5.1 Casos de Testes

Foram selecionados três casos básicos de circuitos lógicos digitais para os testes, são eles: função comparador de quatro entradas, função maioria e paridade ODD (ímpar), ambas de três entradas. Para uma função de três entradas temos uma matriz de ligação de portas NOR composta de dez portas com um total de setenta e cinco células. Nesse caso o cromossomo do GA possui também uma quantidade de setenta e cinco genes. No caso da função comparador por exemplo, que possui quatro entradas, a quantidade de células da matriz e de genes do cromossomo é igual a duzentos e quatro com um total de dezessete portas NOR.

Os parâmetros do GA utilizados para os casos de testes são apresentados na Tabela 5.1. A coluna (n_f) mostra o número de funções (Equação apresentada no Capítulo anterior seção 4.3.1) em relação ao a quantidade de entradas. No caso das funções maioria e paridade que possuem três entradas, a população foi definida com a quantidade de cem indivíduos (n_p), taxa de mutação (m_p) de vinte por cento nos genes do cromossomo filho e setenta e cinco por cento de taxa de *crossover* (c_p), ou seja, setenta e cinco por cento de cem indivíduos foram selecionados para os cruzamentos em pares. Já para o teste do comparador de quatro variáveis, a população (n_p) foi de duzentos e oitenta e nove indivíduos seguindo a Equação 4.2 da população inicial apresentada no Capítulo 4. A taxa de mutação (m_p) e a taxa de *crossover* (c_p) foram as mesmas para esse caso, vinte por cento e setenta e cinco por cento respectivamente.

Ainda neste capítulo, os circuitos minimizados de cada teste são comparados com o melhor resultado encontrado pelo GA. Os testes escolhidos com exceção do "Maioria" fo-

ram os mesmos do trabalho de Barry [27] com o objetivo de fazer uma comparação dos resultados entre os dois trabalhos. Mapa de Karnaugh e McCluskey, por serem métodos semelhantes, apresentam as mesmas expressões booleanas e obviamente os mesmos circuitos lógicos. Por fim, a Tabela 5.2 traz todos os resultados obtidos (medidos em BCs) com o objetivo de comparar os custos de cada circuito com cada um dos métodos de minimização.

Tabela 5.1: *Parâmetros utilizados nos testes do GA.*

Num. variáveis	**População (n_p)**	**(n_f)**	**Mutação (m_p)**	**Crossover (c_p)**
3	100	256	20%	75%
4	289	65,536	20%	75%

Tabela 5.2: *Custos de todos os testes em BCs.*

Custos em BCs				
		GA	**Barry**	**Karnaugh e McCluskey**
Função Maioria	Custo BC	13	-	13
	Qtd de portas	4	-	4
Função Paridade ODD	Custo BC	27	24	33
	Qtd de portas	7	8	11
Função Comparador	Custo BC	41	20	23
	Qtd de portas	12	7	8

5.1.1 Função Maioria

A função maioria de três entradas possui saída '1' se no mínimo duas das três entradas forem '1' de acordo com a Tabela 5.3.

Tabela 5.3: *Tabela verdade para a função Maioria.*

A	B	C	maioria
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Por ser um algoritmo estocástico o GA pode apresentar bons resultados em uma determinada execução e resultados mais ruins em outra. A Figura 5.1 mostra o gráfico para as cinco melhores execuções da função. Observa-se uma queda abrupta do fitness logo nas primeiras gerações. Isso ocorre devido a mudança de etapas, da primeira evolução para a segunda quando os indivíduos passam a ser factíveis e livres dos custos de penalidade. Vale ressaltar que cada valor no plano cartesiano representa o fitness do melhor indivíduo em cada geração.

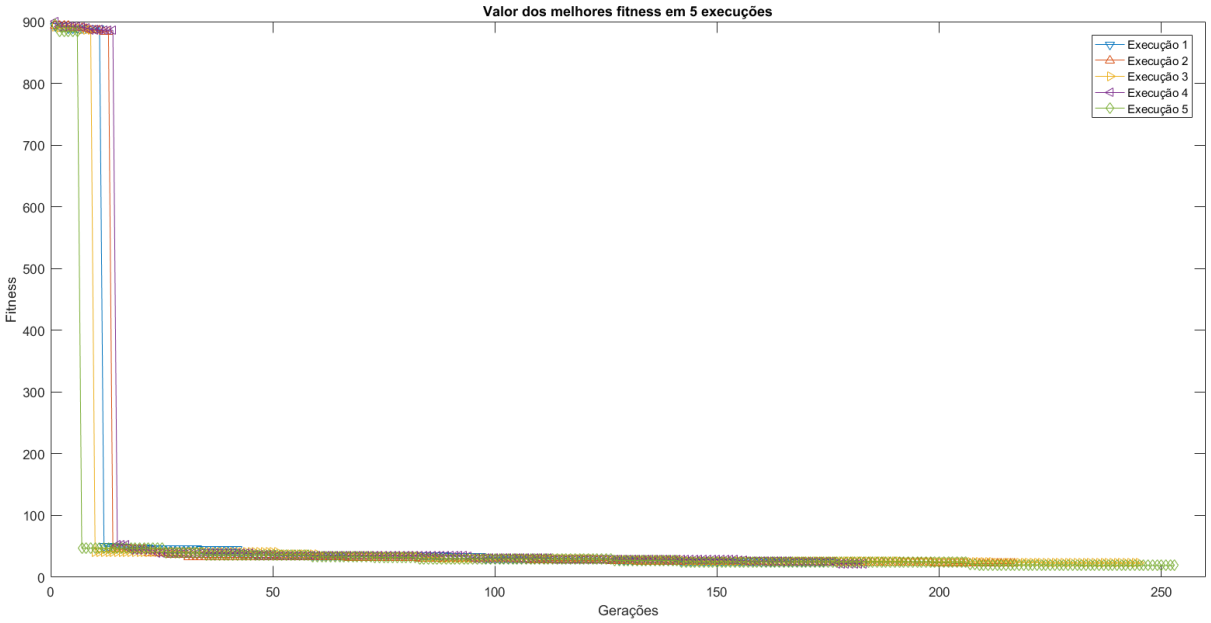


Figura 5.1: *Gráfico melhores de 5 execuções da função maioria.*

Em um pouco mais de duzentos e cinquenta gerações nesse exemplo, o critério de parada foi satisfeito e o algoritmo finalizou a execução sinalizando que o valor do fitness do melhor indivíduo foi menor que o fitness do indivíduo que ocupava então a posição central do vetor população (definição do critério de parada da seção 4.3.6 do Capítulo 4). A partir da segunda etapa de evolução, a tendencia de queda do custo do indivíduo foi suave, se aproximando cada vez mais de um custo mínimo (melhor matriz de ligação possível) da função a cada geração.

Já no gráfico das cinco piores execuções da Figura 5.2 nota-se uma demora maior no número de gerações para se obter todos os indivíduos da população factíveis na primeira evolução do algoritmo. A similaridade das funções das cinco execuções no gráfico mostra que os custos de fitness para um determinado circuito possuem os mesmos níveis de valores a medida que evoluem. Mesmo para esses piores casos, pode-se observar que ao

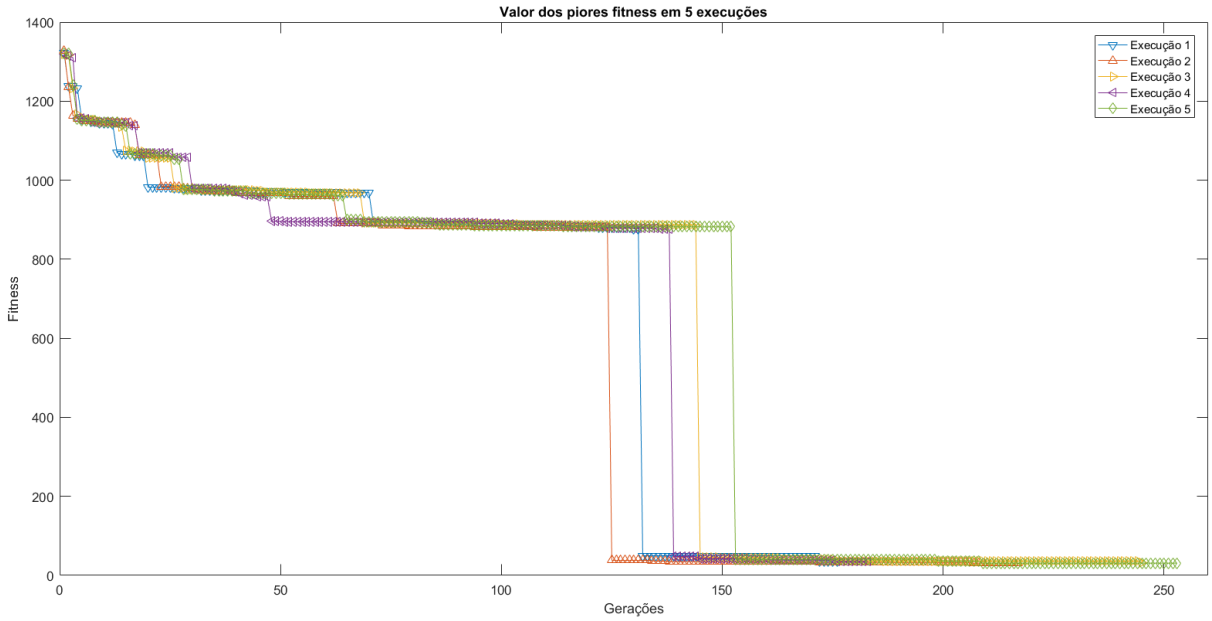


Figura 5.2: *Gráfico das cinco piores execuções da função maioria*

final da execução os resultados são próximos dos resultados finais das execuções em que a população já possuía ótimos indivíduos desde sua primeira geração.

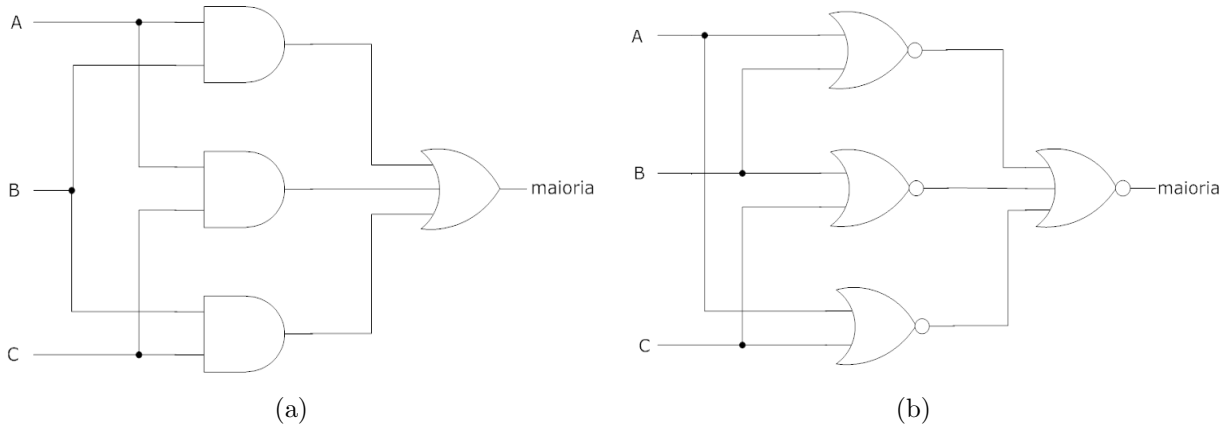
Tabela 5.4: *Testes da função Maioria.*

Execução	Melhor Fitness	Pior Fitness	Gerações
1	18	30	4162
2	17	30	4433
3	19	31	3018
4	19	34	2820
5	20	32	3332
6	18	27	9309
7	**13**	**34**	**3266**
8	19	30	5399
9	17	30	4575
10	19	35	2167
Média	17,9	31,3	4248,1
Desvio Padrão	1,868	2,326	1917,631

A Tabela 5.4 apresenta os valores dos melhores e piores fitness de cada execução e a quantidade de gerações que foram necessárias para a condição de parada do algoritmo ser satisfeita. Para esse caso, a média dos melhores fitness em dez execuções é de 17,9 por cento, uma média de gerações de mais de quatro mil e desvio padrão de 1,4.

A expressão booleana resultante obtida pelos métodos tabulares e pelo GA para a função maioria, é dada por $BC + AC + AB$ e $\overline{(\,\overline{(A+B)} + \overline{(B+C)} + \overline{(A+C)})}$ respec-

tivamente. A Figura 5.3 apresenta os diagramas lógicos para cada um dos resultados (a) e (b). Mesmo que os circuitos possuem diferentes tipos de portas, a quantidade de *fanin* em cada uma delas é a mesma, totalizando um custo de 13 BCs de acordo como mostra a Tabela 5.2.



(a) (b)

Figura 5.3: *Circuito minimizado por: (a) Karnaugh, McCluskey e (b) GA.*

5.1.2 Circuito de Paridade ODD

A Tabela 5.5 mostra o comportamento da função de paridade ODD, que define a saída como '1' quando o número de bits '1' é ímpar.

Tabela 5.5: *Tabela verdade da função paridade ímpar*

A	B	C	Paridade ODD
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

Quando uma população inicial é gerada com indivíduos muito bons, a tendencia é que a primeira etapa de evolução do GA demore menos tempo, melhora os fitness da população mais rapidamente e a execução é concluída com menos gerações. A "execução 1" (azul) no gráfico da Figura 5.4, por exemplo, foi a primeira a iniciar a segunda etapa de evolução e a primeira a ser finalizada, enquanto que a "execução 4" foi o contrário, sendo esta a penúltima a iniciar a segunda etapa e a última a finalizá-la.

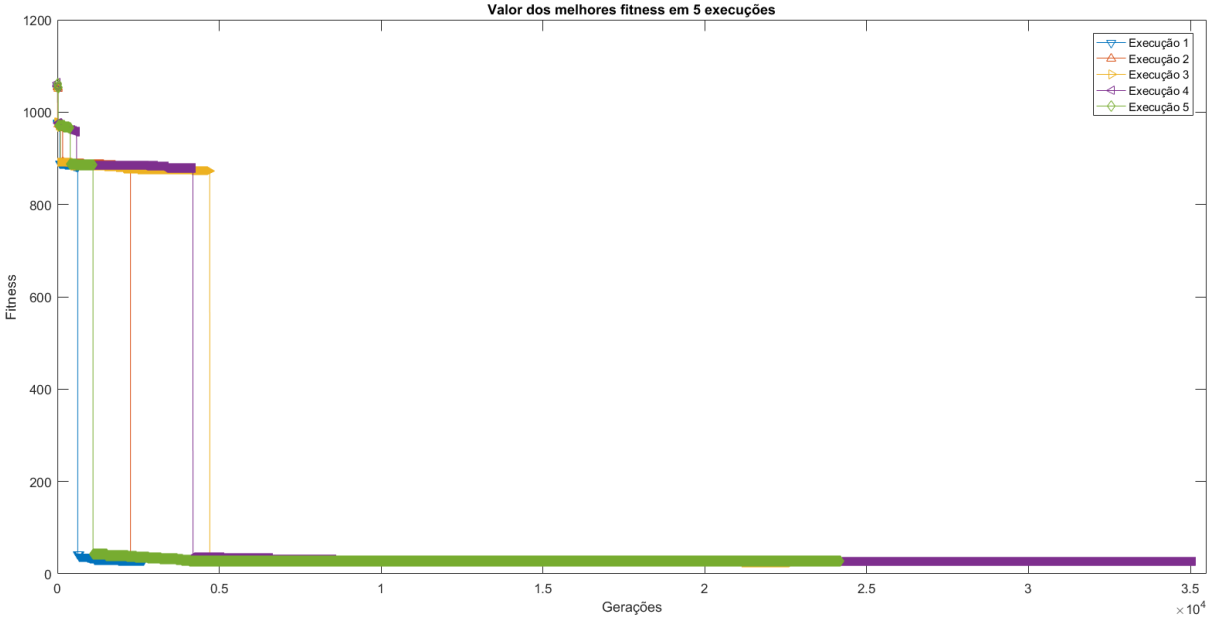


Figura 5.4: *Gráfico das cinco execuções de melhores indivíduos da função paridade.*

As cinco piores execuções para a função paridade ODD, estão representadas no gráfico da Figura 5.5. Note que a "execução 1" (azul) mesmo estando nessa lista foi um teste considerado tão bom quanto alguns dos melhores mostradas na Figura 5.4.

Nos dois gráficos é possível observar que o comportamento da evolução do fitness na segunda etapa, não sofre praticamente grandes alterações ao longo das gerações. Podemos concluir nesses casos que provavelmente o algoritmo encontra, na maioria das execuções, o indivíduo mais otimizado possível para a função.

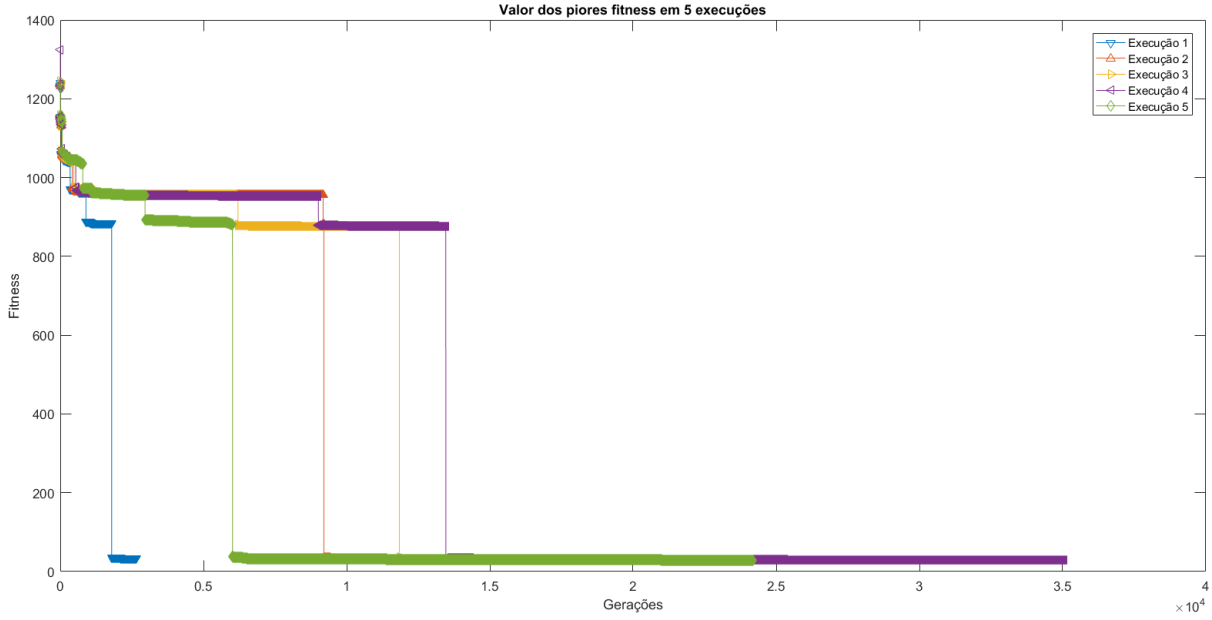


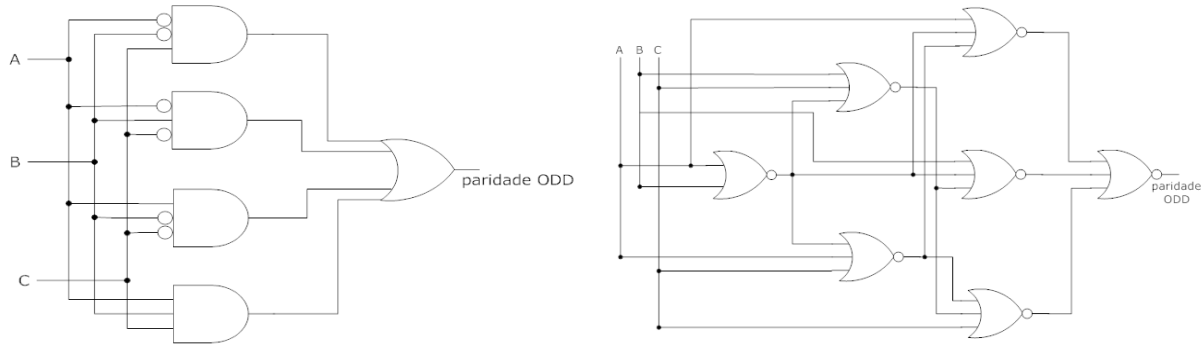
Figura 5.5: *Gráfico das cinco execuções de piores indivíduos da função paridade.*

Outros dez testes realizados são apresentados na Tabela 5.6. O fitness do melhor indivíduo obtido nesse conjunto de testes foi de 24, um pouco abaixo da média de 26,7. O desvio padrão para esse caso foi de 1,487, enquanto que a média de gerações foi superior a treze mil, um valor alto em relação a função maioria que possui o mesmo número de variáveis. Isto deve-se ao fato de que houve uma variação considerável na quantidade de gerações entre as execuções.

Tabela 5.6: *Paridade ODD.*

Execução	Melhor Fitness	Pior fitness	Gerações
1	28	33	2657
2	**27**	**31**	**4987**
3	27	29	18295
4	28	30	22670
5	27	32	4366
6	27	29	35065
7	27	32	9128
8	27	29	24184
9	27	29	7528
10	29	35	4946
Média	27,4	30,9	13382,6
Desvio Padrão	1,487	1,972	10428,624

Com um custo de 33 *basic cells* (BCs) o resultado da Figura 5.6(a) com onze portas lógicas é mais caro que o gerado pelo GA da Figura 5.6(b) com apenas 7 portas e custo de 27 BCs. Entre os testes, a função paridade *ODD* foi a que o GA mostrou ser mais eficiente.



(a) (b)

Figura 5.6: *Circuito minimizado por: (a) Karnaugh, McCluskey e (b) GA.*

Importante observar que a função de paridade ímpar é particularmente difícil de mini-

mizar pelo método de Karnaugh. Isto pode ser visto quando se visualiza a função plotada em um mapa de Karnaugh. O padrão xadrez com a mesma quantidade de uns e zeros, não tem termos adjacentes que podem ser agrupados para a minimização, de acordo com Barry [27]. Cada mintermo da função de paridade ímpar é, portanto, um implicante primo essencial da função, e por isso os mintermos não sofrem desfalque, deixando assim a expressão booleana maior.

5.1.3 Comparador de Magnitude

O comparador de magnitude é uma função aritmética que compara as magnitudes de dois inteiros binários na forma $< i_1, i_2 >$ e $< i_3, i_4 >$. A saída da função tem resultado '1' quando $(i_1.2)+(i_2.1) > (i_3.2+i_4.1)$ [27]. A Tabela verdade 5.7 apresenta o comportamento dessa função para cada combinação de bits.

Tabela 5.7: *Tabela verdade da função comparador.*

A	B	>	C	D	Comparador
0	0		0	0	0
0	0		0	1	0
0	0		1	0	0
0	0		1	1	0
0	1		0	0	1
0	1		0	1	0
0	1		1	0	0
0	1		1	1	0
1	0		0	0	1
1	0		0	1	1
1	0		1	0	0
1	0		1	1	0
1	1		0	0	1
1	1		0	1	1
1	1		1	0	1
1	1		1	1	0

Nesse exemplo de quatro variáveis, a população inicial gerada obteve valores de fitness altos, e pouco menos de vinte e duas mil gerações. Por exemplo, a "execução 2" nesse teste obteve uma população inicial com valores próximos a sete mil de fitness. Assim como nos outros casos de testes, os gráficos das melhores e piores de cinco execuções das Figuras 5.7 5.8 respectivamente, mantém um padrão semelhante entre os testes, quando o algoritmo gera bons indivíduos aleatoriamente na população inicial e quando não gera.

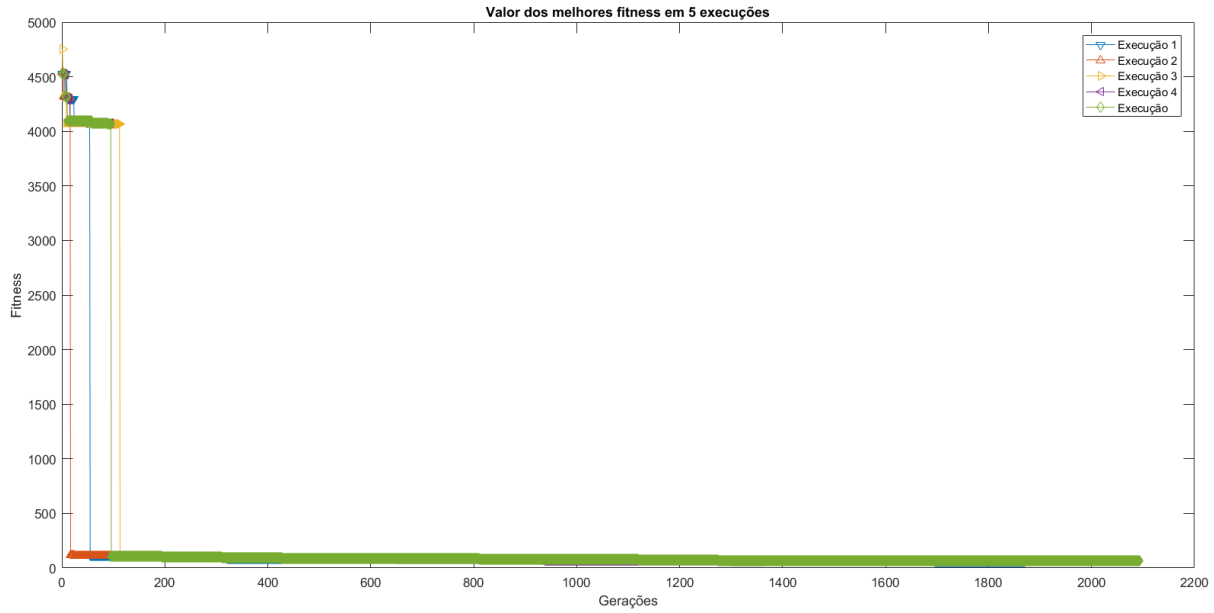


Figura 5.7: *Gráfico das cinco execuções de melhores indivíduos da função comparador.*

Em decorrência do número de entradas desse exemplo, o cromossomo dos indivíduos possuem um número de duzentos e quatro genes, razão pela qual contribuiu para uma maior demora na execução do algoritmo em relação aos outros dois casos de teste apresentados nesse capítulo.

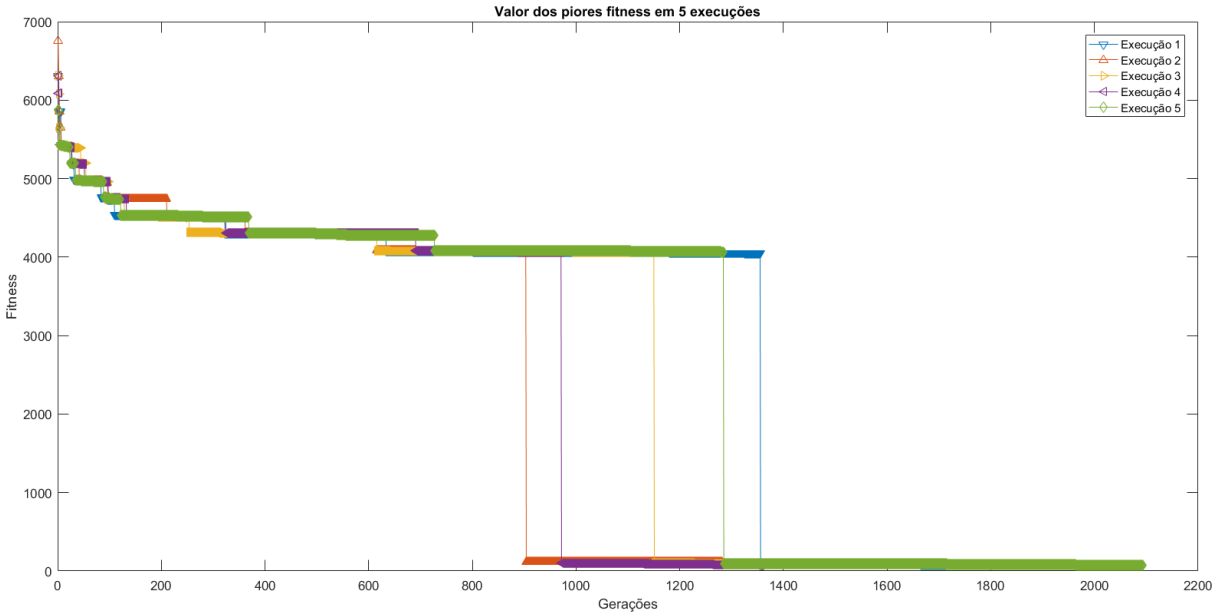


Figura 5.8: *Gráfico das cinco execuções de piores indivíduos da função comparador*

A média de fitness para esta função em relação aos testes da Tabela 5.8 foi de 59,9, com desvio padrão de 4,989 e uma média de gerações de mais de duas mil.

Os testes do AG para a função comparador foram ruins em relação ao resultado obtido

Tabela 5.8: *Resultado de dez execuções da função comparador.*

Execução	Melhor Fitness	Pior Fitness	Gerações
1	56	67	3223
2	60	76	1869
3	**41**	**75**	**2093**
4	66	86	1513
5	71	93	1939
6	57	69	2009
7	57	75	1357
8	54	90	2190
9	57	69	1442
10	58	71	2604
Média	59,9	77,1	2023,9
Desvio Padrão	4,989	8,826	536,892

pelo mapa de Karnaugh e Quine-McCluskey. De acordo com a Tabela 5.2, os métodos tabulares obtiveram um custo de 23 BCs resultando no circuito de apenas quatro portas lógicas da Figura 5.9.

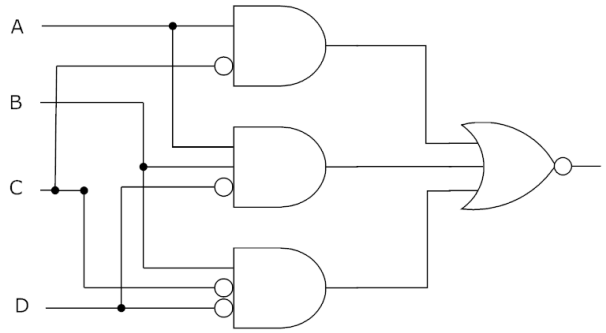


Figura 5.9: *Diagrama do circuito resultante do mapa de karnaugh.*

Já o resultado do GA obteve em sua melhor execução entre as realizadas, um custo de 41 BCs. O diagrama lógico do circuito de 12 portas é mostrado na Figura 5.10.

Nesse caso, não houve um resultado satisfatório do GA, e a razão disto pode estar relacionada à quantidade de execuções ou mesmo à configuração do modelo do critério de parada.

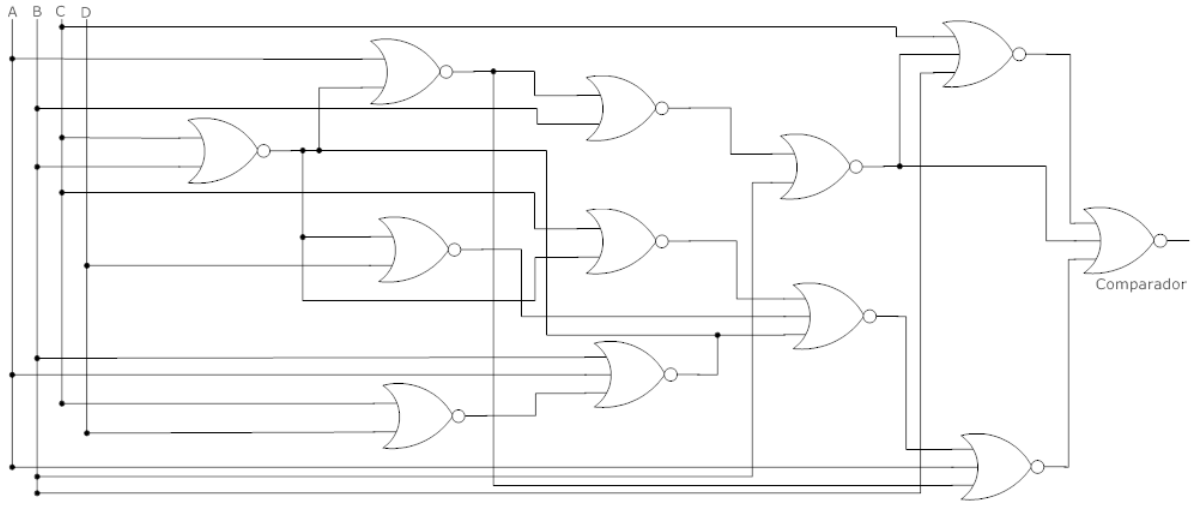


Figura 5.10: *Diagrama do circuito resultante do GA.*

5.2 Considerações finais sobre o capítulo

Os primeiros testes realizados com o Algoritmo Genético determinaram algumas mudanças nas configurações do GA ao longo do desenvolvimento. Uma das alterações feitas, foi a definição do critério de parada apresentado no capítulo anterior. Os resultados obtidos com essas observações permitiram concluir que uma configuração não adequada para o problema pode comprometer a eficiência do Algoritmo Genético e consequentemente a obtenção de bons resultados.

Os resultados obtidos durante o desenvolvimento do trabalho incluindo os apresentados nesse capítulo, mostraram que o GA pode conseguir na maioria dos casos melhores resultados na tarefa de minimização de circuitos combinacionais em relação aos métodos tabulares tradicionais descritos ao longo do trabalho, mesmo com essa desvantagem de possuir um processamento lento. Por essa razão, os casos de testes selecionados precisaram ser mais básicos e possuir um número menor de variáveis.

6 Conclusões

Procurou-se com esse trabalho, demonstrar conceitos, métodos e soluções que possuem relação com o tema, oriundas de outros projetos de pesquisa que serviram de embasamento e auxiliaram na tarefa de desenvolvimento deste com o objetivo de avaliar, comparar, responder questões sobre a possibilidade de otimização de circuitos digitais por meio de técnicas computacionais evolutivas, estocásticas, como o Algoritmo Genético, além de apresentar suas vantagens e desvantagens, limitações e possibilidades que podem ser mais exploradas, estudadas, para obtenção de melhores resultados no futuro.

Ao longo do desenvolvimento deste trabalho foi possível observar que existem uma gama de possibilidades, modelos, métodos e técnicas que podem ser adotadas e testadas em relação a esse tema. Conforme discutido em outros Capítulos utilizou-se do projeto intitulado de "Síntese de Redes Lógicas Multiníveis de Custo Mínimo Via Algoritmo Genético", ou seja, Síntese de redes lógicas multiníveis de custo mínimo por Algoritmo Genético, em sua forma traduzida, como base para a definição das técnicas utilizadas, como por exemplo: o cálculo de fitness, a codificação do cromossomo, seleção dos pais, e método de avaliação. Esse foi o artigo base deste trabalho, o que possibilitou ter um conhecimento prévio sobre as desvantagens da utilização do Algoritmo Genético para esse tipo de problema e em contrapartida os resultados positivos que foram obtidos.

Alguns operadores genéticos definidos anteriormente foram descartados ao longo do processo de construção do algoritmo a medida que se obtinha maior clareza em relação a problemática, procurando assim um caminho mais simples para evitar uma maior sobrecarga de operações.

Importante ressaltar que foram encontradas também algumas dificuldades com relação à definição do critério de parada em alguns modelos testados, desde a parametrização do tamanho da população que automaticamente dispensava o critério de parada, até o modelo final apresentado no Capítulo 4. Observou-se também que, dependendo de como foi definido, o critério de parada influenciou no tempo de execução do algoritmo, determinando assim uma média de quantidade de gerações que o algoritmo levaria para apresentar a solução.

Outro ponto importante observado, foi em relação ao modelo que define o tamanho

da matriz de conexões e consequentemente o cromossomo. Apesar de ser bastante útil, simples, e resolver perfeitamente o problema, um simples circuito de quatro entradas torna o vetor de genes e o número de células da matriz de conexões bastante grande aumentando assim a complexidade do algoritmo, principalmente quando, por exemplo, a operação do cálculo da saída da função (tabela verdade) é feita para cada indivíduo.

A principal desvantagem da abordagem baseada em Algoritmo Genético é seu fraco desempenho de tempo de processamento, principalmente se for executado em computadores convencionais, o que foi o caso desse trabalho, para o qual utilizou-se dessa arquitetura por falta de infraestrutura necessária. Por outro lado, a vantagem dessa abordagem é que os resultados são superiores em termos de custo de circuitos quando comparados a métodos tabulares apresentados nos Capítulos anteriores. Uma vez superada a principal desvantagem mencionada, o método torna-se uma excelente ferramenta na tarefa de minimização e pode ser utilizada de forma mais ampla.

Referências Bibliográficas

[1] GAJSKI, D. D.; KUHN, R. H. Guest editors' introduction new vlsi tools. *IEEE Computer*, p. 11–14, 1983.

[2] RIESGO, Y. T. T.; TORRE, E. de la. Design methodologies based on hardware description languages. *IEEE Transactions on Industrial Electronics*, p. 3–12, 1999.

[3] GERSTLAUER C. HAUBELT, A. D. P. T. P. S. D. D. G. A.; TEICH, J. Eletronic system-level synthesis methodologies. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, p. 24–33, 1975.

[4] CHUNG, Y. T.; JIANG, J. H. R. Functional timing analysis made fast and general. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, p. 1421–1434, 2013.

[5] KENG, B.; VENERIS, A. Path-directed abstraction and refinement for sat-based design debbugging. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, p. 1609–1622, 2013.

[6] DAMAVANDPEYMA S. STUIJK, T. B. M. G. M.; CORPORAAL, H. Schedule-extended synchronous dataflow graphs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, p. 1495–1508, 2013.

[7] MORVAN, S. D. A.; QUINTON, P. Polyhedral bubble insertion: a method to improve nested loop pipelining for high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, p. 339–352, 2013.

[8] GONG S. BASIR-KAZERUNI, L. H. F.; YU, H. Stochastic behavioral modeling and analysis for analog / mixed-signal circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, p. 1517–1530, 2009.

[9] ALMEIDA, T. da S.; SILVA, A. C. R. da; GROUNT, I. A. Methodology analisys of a computational tool used in eletronic circuit design. In: . [S.l.]: IEEE Conference Publications, 2011.

[10] ERDENER, E. G. *CAD Standards and Institutions of higher education*. [S.l.]: Facilities, 2001.

[11] HYDE, R. *The Art Of Assembly Language*. [S.l.]: no starch press, 2001.

[12] TOCCI, N. W. R.; MOSS, G. *Sistemas digitais: princípios e aplicações*. 10. ed. [S.l.]: Person, 2008.

[13] OLIVEIRA, V. C. de. Projeto e otimização de circuitos digitais por técnicas de evolução artificial. *UnB*, 2015.

[14] MORENO FABIO PEREIRA, C. P. R. P. E. *Projeto, Desempenho e Aplicações de Sistemas Digitais em circuitos Programáveis (FPGASs)*. 1st edition. ed. [S.l.]: Bless, 2003.

[15] TARNOFF, D. *Computer Organization and Design Fundamentals*. 1st edition. ed. [S.l.]: Lulu, 2007.

[16] AGARWAL, A.; LANG, J. H. *Foundations of Analog and Digital Eletronic Circuits*. 1st edition. ed. [S.l.]: Denise E. M. Penrose, 2005.

[17] MANO, M. M. *Digital Logic and Computer Design*. [S.l.]: Prentice Hall of India, 2004.

[18] NELSON H. TROY NAGLE, J. D. I. V. P.; CARROLL, B. D. *Digital Logic Circuit Analysis e Design*. 1st edition. ed. [S.l.]: Prentice-Hall, 1995.

[19] DANDAMUDI, S. P. *Fundamentals of Computer Organization and Design*. [S.l.]: Springer, 2003. 67 p.

[20] GIVONE, D. D. *Digital Principles and Design*. [S.l.]: McGraw-Hill Professional, 2003. 173 p.

[21] JAIN D. S. KUSHWABA, A. K. M. T. K. *Optimization of The Quine-McCluskey Method for the Minimization of the Boolean Expressions*. [S.l.: s.n.], 2008. 2-4 p.

[22] BENTO, E. P.; KAGAN, N. *Algorítmos Genéticos e Variantes na Solução de Problemas de Configuração de Redes de Distribuição*. [S.l.]: Controle & Automação, 2008.

[23] E., G. D. *Genetic Algorithms in Search, Optimization and Machine Learning*. [S.l.]: Reading, MA, 1989.

[24] COELLO ALLAN D. CHRISTIANSEN, A. H. A. C. A. Towards automated evolutionary design of combinational circuits. *211 Stanley Hall Departament of Computer Science*, 2001.

[25] LOUIS, S. J.; RAWLINS, G. J. Using genetic algorithms to design structures. *Technical Report*, Computer Science Department, Indiana University.

[26] COELLO, C. A. An empirical study of evolutionary techniques for multiobjective optimization in engineering design. *Phd*.

[27] SHACKLEFORD ETSUKO OKUSH, M. Y. H. K. B. Synthesis of minimum-cost multilevel logic networks via genetic algorithm. *Sasimi*, 2000.

[28] LANCASTER, D. *TTL Cookbook*. 23nd. ed. [S.l.]: SAMS, 1992.

[29] Synopsys Design Compiler. <http://www.synopsys.com/Tools/Implementation/RTLSynthesis/DesignCompiler/Pages/default.aspx>. Accessed: 2016-11-21.

[30] SOBRINHO, S. C. A. M. E. F. G. Ehw aplicado a síntese de circuitos digitais usando representação por portas lógicas. *Simpósio Brasileiro de Pesquisa Operacional*, 2006.

[31] S LACERDA, B. d. A. S. C. F. M. T. W. Síntese de circuitos digitais utilizando computação evolutiva. *Ufla*, 2010.